A Graphical Relational Query Language On Apple Macintosh

A thesis

submitted in partial fulfillment

of the requirements for the Degree

of

Master of Science in Computer Science

in the

University of Canterbury

by

M.Naguleswaran

University of Canterbury

1988

Acknowledgements

I wish to thank Dr. Neville Churcher for supervising this project. The help, encouragement and the time he freely gave from the initial stages through to the final preparation of this thesis is gratefully acknowledged.

The Battersby scholarship awarded to me by Datacom Systems Ltd. for the year 1988 is much appreciated.

I thank James Collier for the help he gave in putting together the Unix/Macintosh communication routines. Thanks are also due to Yalini Sundralingam for proof reading the final draft.

Abstract

Graphical presentation has been used effectively to reduce complexity long before computers were invented. The Macintosh's user interface is the motivation for this project to design a graphical relational query language to facilitate easy querying. A graphical methodology for expressing queries has been developed and implemented.

This thesis explores the pros and cons of alternative approaches for graphical expression of queries and explains the basis for the design of GQL and gives a description of the GQL system itself.

In addition to the Macintosh's user interface as an easy to use interaction medium, the GQL system that has been implemented provides further user aids to query formulation. This is achieved by maintaining a local dictionary in which the access path information and predefined subqueries can be permanently stored and used in the formulation of new queries. Another notable feature of GQL is the modular definition of a query, where the module detail can be viewed or hidden using graphical techniques. The design of GQL permits it to be interfaced with any relational DBMS with minimum effort.

Table of Contents

Introduction

An end user sitting in front of a terminal attempting to retrieve information stored in a computer is becoming a common scenario. The developments in technology that have lead to the above scenario are as wide as the developments in the computer field itself. The major contribution is made by the availability of less and less expensive computers, which in turn has spurred the need for improved user interfaces for users of wide ranging backgrounds.

The database system to which the user is directing the query has evolved rapidly over the past twenty five years in its own right. Beginning in simple programs that handled formatted files, information storage and retrieval has evolved through hierarchical database management systems which is a model abstraction imposed on the IBM product IMS [Date 86] and the network model based on the CODASYL [DBTG71] proposal. The relational model [Codd70] represents the current state of database technology with a lot of research interest directed towards many areas related to it [Schi88]. The user interface to relational databases is one of them.

The interface employed between the end user and the rest of a DBMS is very varied in its mode of interaction as well as in the method of implementation. Modes of interaction employed in these interfaces range from simple function key driven interfaces to complex menu driven types. These are usually generated as application programs. The relational model proposed included relational query languages [Codd72] for use as data manipulation statements embedded in a standard programming language. With this approach application specific interfaces can be developed. To improve productivity of application generation, fourth generation tools [Mart85b] are employed. Fourth generation tools essentially provide for rapid generation of applications by providing for definition of screen layouts and access and control of these screens as well as the data manipulation ability.

Relational query languages as directly usable systems paved the way for the next stage in interface development. This approach contrasts with the application generation approach in avoiding an intermediary to develop the application to be used by the end user. The text based languages that were provided as directly usable interfaces were found to be difficult for general use [Reis77].

To provide an easier to use interface along this line is attempted in two ways. These are natural language interfaces and graphical interfaces. In the natural language interfaces [Codd78] the user is permitted to type queries in natural English. While natural language querying is useful for answering ad hoc queries, graphical querying can be further enhanced to incorporate additional features as mentioned below in an interface that is application independent. End user functionalities that are provided through these database interfaces include

1. Querying
2. Updating
3. Browsing (a special mode of querying)
4. Database editing (a special mode of editing)
5. Report generation

Various end user interface generation approaches are depicted in figure I.1. Path A is the traditional one that requires a professional application developer. GQL is an attempt to develop a user friendly query language in path B. GQL can also be used to generate a user interface along the approach marked 'C' by which a query translated from a GQL expression can be used within a host system development tool to generate a user interface.

In chapter 1, some of the concepts involved in a graphical interface to relational databases are introduced. Chapter 2 covers the relevant areas of relational technology for implementing a graphical interface. These include the techniques that have already been employed for expressing relational queries graphically and the results of human

factor studies over some of these techniques. In chapter 3 a basis for graphical query expression is developed and its selective power is analysed in detail. This is used to derive a classification that was useful in the implementation of GQL.

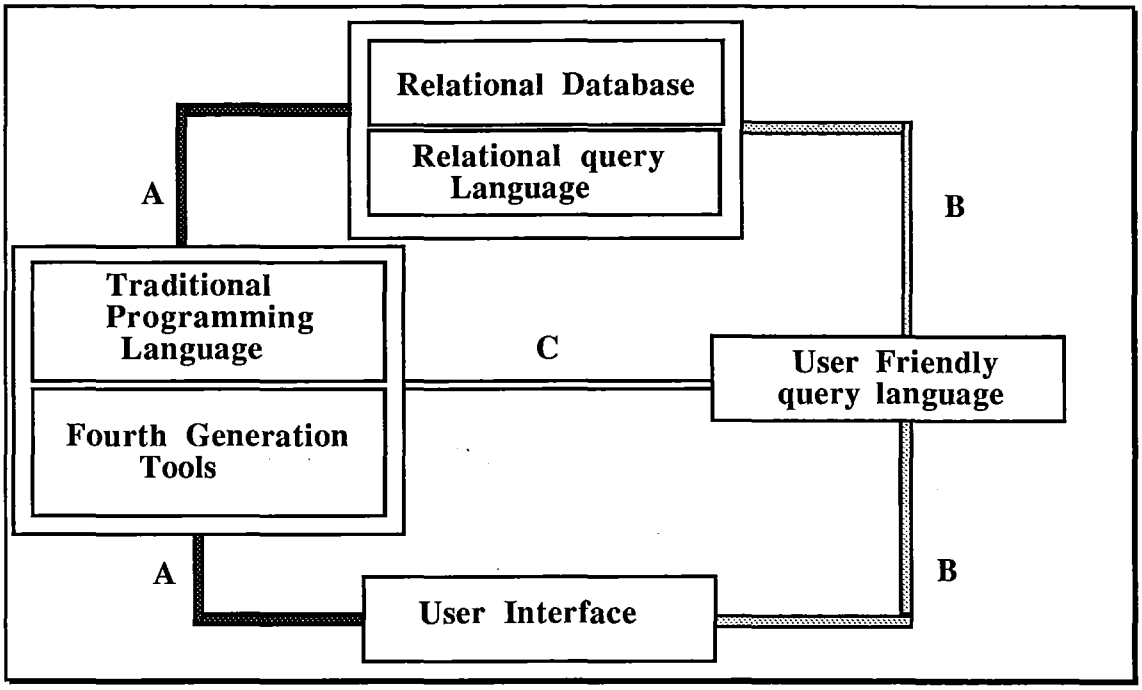


Fig I.1

Another area that contributes to the user interface design is the work done in semantic modelling techniques [Hull87]. The original proposal for relational database and its early implementations were mainly aimed at preserving data independence and providing for a query language to apply the required semantics. The relational model itself contains almost no semantic information stored as part of its schema definition. Thus the responsibility is with the user of a relational query language to specify the semantics as part of the query. One advantage of semantic modelling is that the additional information representing the meaning of the database can be used to develop a more intelligent interface thus relieving the user of the query language from this task. Techniques in semantic modelling and some query languages based on one of these models are described in chapter 4. Some conclusions are reached at the end of chapter

4 as to the technique to be used in GQL. GQL maintains a local dictionary in which additional information about the database can be stored and used through the graphical interface for the purposes of query building and query translation. Chapter 6 presents a high level view of the proposed solution by detailing the local dictionary structure of the GQL system.

GQL uses the Apple Macintosh interface for querying. The Macintosh interface provides a more widely available yet powerful front end for implementing GQL. Any relational DBMS that can execute a QUEL (§2.3.2) query and includes a serial line port can be interfaced to GQL.

A query is built by selecting objects with the mouse and issuing commands using the pull down menus. Relevant entities, attributes, relationships and predefined subqueries in the local dictionary are presented as scrollable lists on the users request, for building a query. Selecting objects from a legal list of alternatives reduces the error probability during query building. GQL permits building and viewing queries in a modular fashion. Subqueries can be defined and used in other queries. This approach, in conjunction with graphical techniques to expand and condense objects, provides a powerful method for reducing the perceived complexity of a query by the user. User friendliness of an interface as an absolute quantity cannot be precisely defined. Chapter 5 describes in detail the graphical techniques used by GQL, to build the dictionary and for querying the database. This chapter can be treated as an indication of the userfriendliness of GQL.

Figure I.2 shows the GQL form of the query '*Get item names and sale dates of those sales handled by employee Tim Jones from those departments which are managed by some one other than Tim Jones' manager*'. Some of the query objects have been expanded in the figure. The 'ViewWindow' in the figure shows the result of this query.

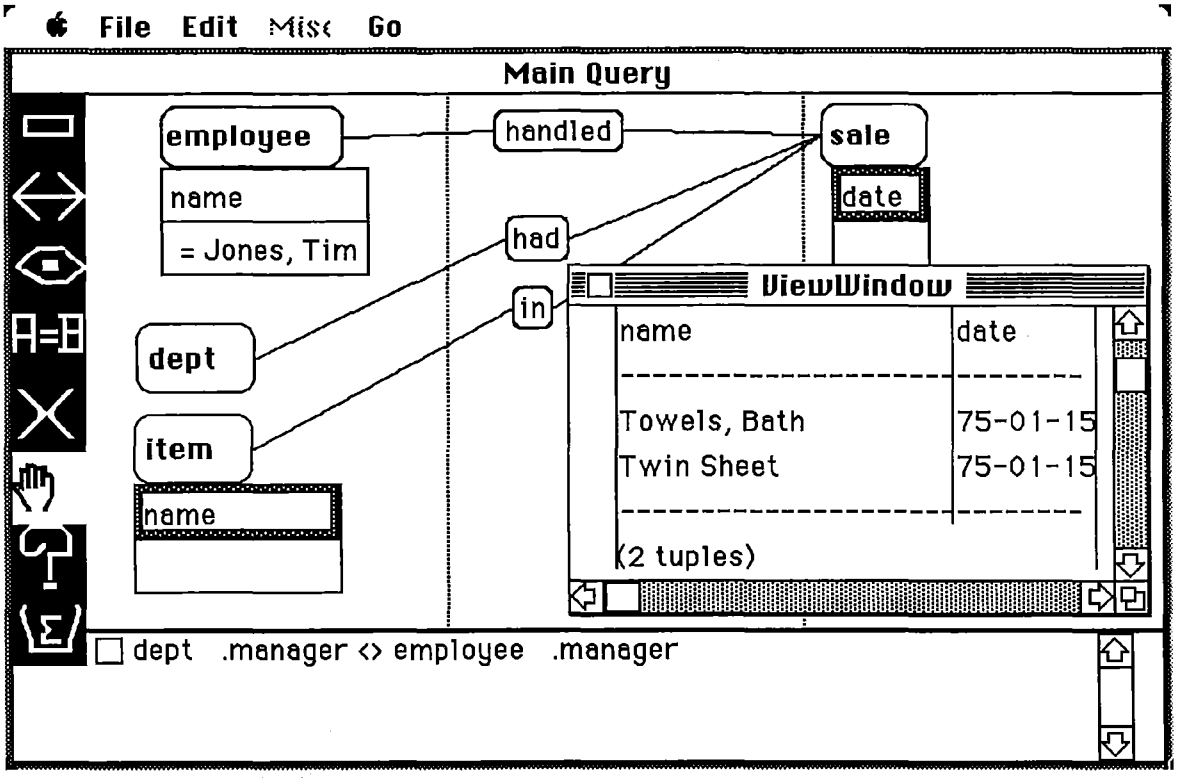


Fig I.2

The equivalent query in QUEL is given below. The complexity reduction as well as ease of query formulation achieved by GQL can be assessed by comparing these two forms for expressing the same query.

```
Range of item is item
Range of emp is employee
Range of sale is sale
Range of dept is dept
Retrieve ( item.name, sale.date )
Where
 emp.name = "Jones, Tim"
 AND item.number = sale.item
 AND emp.number = sale.employee
 AND sale.dept = dept.number
 AND sale.store = dept.store
 AND emp.manager != dept.manager
```

Some further design and implementation issues are summarised in chapter 7. Notably the translation algorithm is given in this chapter. A query is maintained in an internal structure by GQL and the current version includes routines to translate it to QUEL queries.

The aim of this project has been to develop an interface that can be adapted to any relational database system. Using the interface some of the semantic information about the host schema can be built into the GQL dictionary which is resident at the Macintosh end. Thus the host system dependent part of the GQL system must be isolated so that the effort required to adopt it to a particular system is minimised. This part of the GQL system is described in chapter 8.

Chapter 1

Issues In Graphical Interfaces

1.1 Introduction

People use language as a mode of communication. The thinking processes of people are affected by the language that is used for thinking[Mart85a]. Jarke [Jark85] claims that the number of syntactic constructs to be remembered by a user and the dynamics of how the user is allowed to formulate commands are aspects of the language that influence the thinking effort required. Thus the design of any tool for communication purposes should accommodate both the issue of communication and its aid in the thinking process.

When computers are utilised to perform complex tasks the need for good communication languages is even more acute. As the complexity of computer applications increases, while the expertise level of its users decreases, this requirement becomes even more critical. This is evident in the increased interest in Human Computer Interaction research and the different modes of communication that are being tried [Herot82].

Graphical modes of expression have been used by people for communicating complex information among themselves. Certain groups of specialists such as architects and surveyors have developed standards for formal two dimensional communication languages. Graphical computer terminals supported by high computing power can be used to implement these formal languages on a computer. Such applications provide the added ability of easy changes to the diagrams and integrity checking that may be placed on the structure of the diagrams, thus relieving the user from some mental load and increasing the quality of work. With well designed graphical languages the use of graphical terminals can be extended to include the issue of communicating commands to

the computer itself. This is a relatively new field and the development in this area will be driven as has always been the case by the availability of hardware and the interaction techniques. The availability of one usually produces the other.

Shneiderman [Shne82] notes the following reasons for the enthusiasm shown by users of graphical terminals using what he calls 'direct manipulation techniques'.

1. Novices can learn basic functions quickly, usually through a demonstration by a more experienced user.
2. Experts can work extremely rapidly to carry out a wide range of tasks even defining new functions and features.
3. Knowledgeable intermittent users can retain operational concepts.
4. Verbal or printed messages are rarely needed.
5. Users can immediately see if their actions are furthering their goals and if not they can simply change the direction of their activity.
6. Users have reduced anxiety because the system is comprehensible and because their actions are easily reversible.

In [Chan86] visual languages are categorized into following classes.

1. Logical objects with visual representation
2. Visual objects with imposed logical representation

The first type includes techniques such as Nassi-Shneidermann charts and data flow diagrams. The second type includes interfaces to information systems where the information represents physical objects that can be represented as such on the screen.

1.2 Interaction Techniques Available on Current Hardware

Research in human factor considerations has produced some good guide-lines for interaction methodology [Gai83]. Systems are available that incorporate these techniques to provide a development environment for producing applications with good graphical user interfaces. The essence of these techniques can be summarized as

Windowing and scrollbars

Pulldown menus

Entering commands as direct actions on objects

Expansion and shrinking of objects to view levels of details

The essence of developing a graphical application is to design a graphical representation of reality or metaphor or paradigm. A good example of such design is the desk top metaphor of the Apple Macintosh system.

1.3 Graphical Interface to Relational Databases

While original query languages were linear languages based on relational calculus or relational algebra or tabular as in QBE [Zlo77], the need for easier to use query languages has generated varying forms of interfaces using graphical terminals. The tabular form of query languages initiated by the QBE example discussed in §2.3.3 proved very popular. With improved graphical terminals and techniques even greater advancement in this direction should be possible. The classification of graphical interfaces to relational databases shown in figure 1.1 is used for discussing various aspects of the graphical interface. The classification is based on the 'degree of graphical representation' of the information concerned.

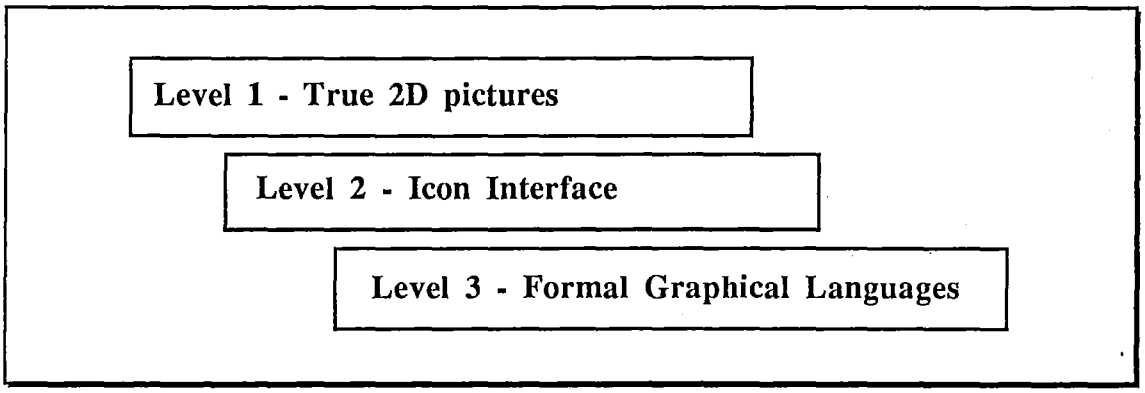


Fig 1.1

1.3.1 True 2D Pictures

In certain applications true 2D pictures of the real word can be represented in the graphical interface to aid the user in expressing commands. For example a section of the picture can be annotated to request more detail or request aggregate properties of the selected portion.

Some examples of such applications are

1. Examples of Computer Assisted Design databases
2. Geographical maps in a mapping system.

In such applications the degree of graphics usage can be said to be high, with the inevitable increase in the computing resources required. Such interfaces are feasible only in some specialized applications.

1.3.2 Iconic Interface

This mode of interface is where the database is represented by icons on the graphical screen which can be directly manipulated by the user. Such an interface requires identifying the objects of interest and designing appropriate icons to represent them spatially on the screen. Such an approach is very much dependent on the information in the database and thus application dependent. It is also recognised that the skills required for the design of icons are not necessarily intuitive. Due to the effort required to design the iconic representation such interfaces are best suited when the database is static and not suitable as a general query interface. Advantages of such an interface to users are however obvious. Such systems are very useful where the users expertise level is expected to be low and the requirements of the interface in regard to its computing power can be precisely defined. Spatial Data Management System [Herot82] and Icon Based Database Management IBMS [Fras86] are two systems that take this approach.

In SDMS in addition to the graphical representation, three screens are used as three frames of reference. The first screen presents a global view of data and more detailed views of selected objects are presented in the the other screens in increasing order of detail. The user is said to be better oriented with these separate frames of reference and better understand exactly where in the database the inquiry has led to. The spatial frame work helps the user to locate information more easily. This technique of viewing details that are organised in a hierarchical structure is employed in GQL to hide and view query details thus improving comprehension (§5.5-§5.10).

1.3.3 Formal Graphical Languages

This is the last category of graphical interface that is identified. The challenge here is to develop graphical techniques to communicate complex information. Such a design should not only develop the graphical formalism to represent information, as those standards that have been developed using paper and pencil tool do, but also include the actions that the user performs on them to communicate commands.

The work presented in this thesis is an effort in this area. As mentioned at the start of this chapter a graphical representation of relational queries has been developed and the thinking effort required to formulate queries is minimised by the visualisation of the logical objects and by the dynamics that allow the query formulation. According to the classification in [Chan86] (§1.1), this type of language uses visual representation of logical objects. Although this type of representation is the major part of such a language, the representation of physical entities in the database such as books or persons as objects in the interface is also possible. A good graphical interface could well incorporate both types of visualisation.

In the following sections some techniques used for graphical expression of conditions are described. Their applicability in GQL is also discussed. In chapter 2 some

existing implementations of relational query languages that use these techniques are discussed and further directions for the GQL implementation are noted.

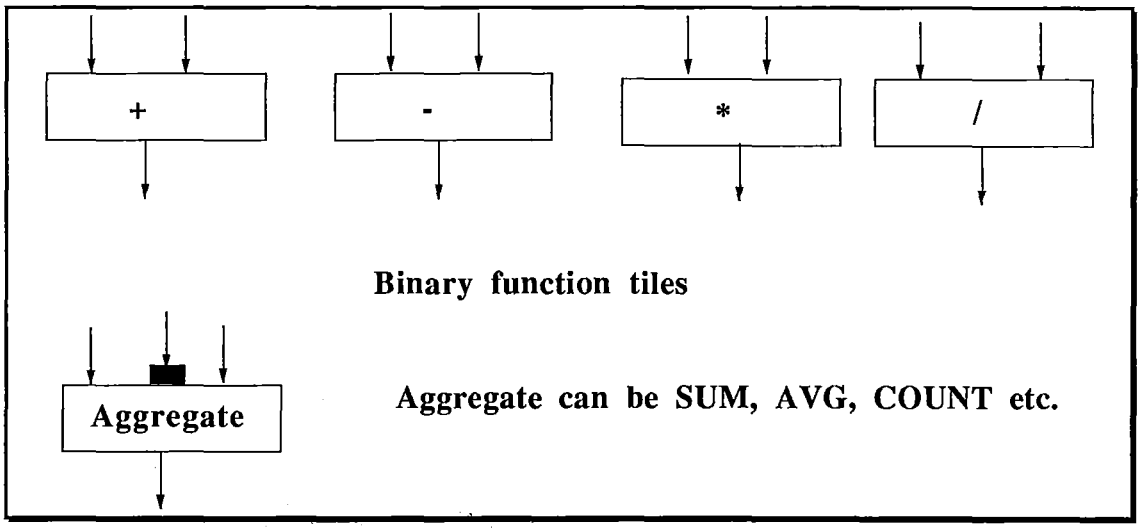


Fig 1.2

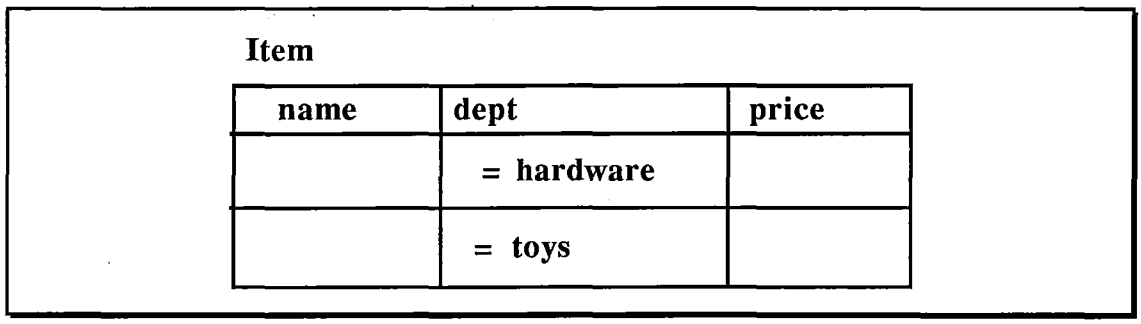
1.3.3.1 Function Tiles Technique

A typical set of function tiles that is useful is shown in figure 1.2. Functions are provided as predefined tiles in the system. Function tiles are dragged into the building area and their operand fields are fed in and result field fed out of the tiles. By connecting many function tiles formed in this manner a complex condition can be generated. PROGRAPH [Mat85] is a graphical programming language that has been proposed. This too uses the technique of function tiles for graphical representation of expressions.

1.3.3.2 Tabular Techniques

In this technique simple scalar comparisons are placed on fields represented as the columns of the table. All conditions placed in a row of the table are implicitly ANDed, while those on separate rows are implicitly ORed. The conditions in figure 1.3 are

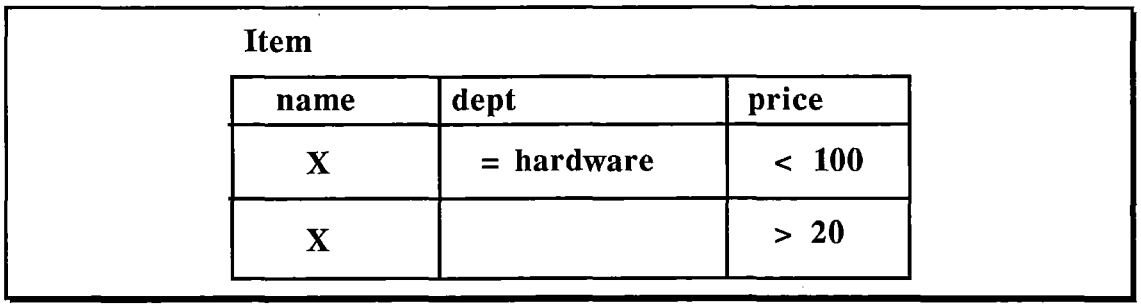
interpreted as *'Items of hardware department or items of toy department.'*. The technique as described above is not powerful enough to express all possible conditions that involve the fields of the table. The condition *'Items of hardware department that cost less than 100 and more than 20"* cannot be expressed in the above technique. One extension to this technique is used in Query By Example [Zlo77] notation discussed in §2.3.3. In this method conditions in separate rows are ANDed instead of ORed if they are linked by a common name in one of the fields as in figure 1.4. Another extension to this technique [Urs83] allows logically ANDed conditions of simple scalar comparisons to be placed on fields as in figure 1.5, thus separate rows are always ORed.



Item

name	dept	price
	= hardware	
	= toys	

Fig 1.3



Item

name	dept	price
X	= hardware	< 100
X		> 20

Fig 1.4

Item

name	dept	price
	= hardware	< 100 AND > 20

Fig 1.5

1.3.3.3 Techniques for GQL

Some form of expressing conditions must be provided in a query language. The method of specifying these conditions must be easy to build as well as easy to read and understand. The power of the method used for specifying conditions also has direct bearing on the ability of the language to express selection conditions.

The function tiles approach mentioned has the advantage of being very powerful. It also facilitates the building of expressions using direct manipulation of objects. The disadvantages are,

1. The space required to construct an expression in terms of the screen space available is relatively high. Thus an expression that can be written in one line in the standard text based format, when constructed and displayed with this technique may have to be viewed by scrolling. This results in comprehension difficulty.
2. This form of display is also not in keeping with the training people have in expressing calculations. One is more used to text based calculations and the bracketing notation.

In GQL the function tiles technique as described is not used for the above reasons. However a simple condition facility that compares the values of two objects is provided which can be built using direct manipulation of the objects (§5.8). The actual display of the condition is in the standard text notation and not in the function tiles notation. This

approach is easy to use for simple conditions that involve two values only. The GQL approach has separated a commonly used feature of relational query languages, namely comparisons of two fields in the database, to apply this technique.

The table concept is an important aspect of relational databases since relations are stored in tables. The results of a retrieval are also presented as a table to the user. Therefore expressing queries in a tabular form is attractive to the user. The tabular technique is also closer to the text based expression and thus may be more natural. GQL also uses the tabular technique where it is appropriate. The short coming of the tabular technique is that it is not as powerful as the text based expression. In a friendly interface it is reasonable to forego some power of expression to gain the userfriendliness. Therefore GQL uses tabular technique of figure 1.5, where it enhances the modularity in the display of queries.

Chapter 2

Relational Language Implementations

2.1 Introduction

Relevant relational query languages are surveyed in this chapter. The text based languages SQL and QUEL are studied since the GQL queries must be translated to an equivalent query in one of these languages for execution by the appropriate DBMS.

In addition this chapter covers graphical relational query languages, namely QBE and CUPID, based on the relational model. The need for complexity management becomes evident when surveying these graphical languages. This is also discussed in this chapter. Results of some human factor studies performed on these languages are listed at the end of this chapter and the GQL approach to address the difficulties faced by the user is noted. Improved interfaces to relational databases have been attempted that assume data models that deviate from the relational model. These approaches to data modelling, referred to as semantic modelling are discussed in chapter 4.

2.2 Relational Model

To quote from [Date 86] chapter 15 pp 313, a relational model is defined as

Data Structures

domains (atomic values)
n-ary relations (attributes, tuples)

Data Integrity

1. primary key values must not be null
2. foreign key values must match primary key values (or must be null)

Data manipulation

relational algebra (or relational calculus equivalents) :
relational assignment

Theoretically this is stated as the minimum requirement. However commercial relational systems available have not come up to these specifications. The current INGRES system does not provide for the definition of domain classes or for the maintenance of the two integrity rules. A point to be noted is the gap between the theoretical developments in relational systems and the available implementations on top of which the user interface I have developed is to operate. Some desirable features that are achievable with a relational system that is based on the current state of the art, had to be ignored, since the GQL system cannot achieve this independent of relational implementations.

2.3 Relational Query Languages

Relational Algebra and Relational Calculus were proposed to define the power of manipulative language that should be provided in a fully relational system. These two classes of languages are used to define the concept of relational completeness for a relational query language [Codd72]. The proposals are aimed not only at the manipulative operations of a database but also as a tool for the administration of the database in defining its security and integrity rules. Some of these implementations are summarized below. The query *Find departments situated in Christchurch and selling items that cost more than 100* is used to illustrate the major features of the implementations.

2.3.1 SQL

The SQL [Date86] language was initially implemented as the DML for the System R project by IBM Research Laboratory [Ast76], and subsequently marketed with its database products. The ANSI-SPARC study group recommendation [Jard77] for a query language also has close resemblance to SQL. Thus it has become a de facto industry standard for relational query languages and used as the common language across different systems.

The SQL language includes constructs from both the algebra and calculus formalism for query languages. Of all the text based relational query languages SQL expresses the logic of the query in the most comprehensible manner using nested subqueries. The subquery structure of SQL can be used to modularise sections of a query. SQL still suffers the common shortcomings of text based query languages as a casual user interface. The query stated in the previous section is expressed in SQL as

```
Select  Dept.Name
From Dept, Store
Where (Dept.number = Store.dept)  AND  (Store.city = Christchurch)
AND
Dept.number IN
      Select Item.dept from Item Where (Item.price > 100)
```

2.3.2 QUEL

QUEL [Sto76], [Date86] is the language of the INGRES relational database system. QUEL is based on the calculus formalism for a query language. QUEL's provision of a single construct for all types of queries at first may appear easy to learn. However the single structure also makes it a difficult language in which to express more complex queries clearly. This is somewhat like the BASIC programming language which is easy to learn and difficult to use. QUEL requires tuple variables to be declared explicitly as in relational calculus [Codd72]. Thus query formulation must proceed in a programming like manner. All the problems attributed to the SQL as a casual user language can also be attributed to QUEL. The absence of nested queries and the lack of explicit quantifiers are additional shortcomings in QUEL. While QUEL too may be suitable as a language for DB professionals due to its expressive power it is not satisfactory as a casual user language. As a machine readable canonical form QUEL 's uniform style is more suitable than SQL. QUEL's BY clause too is more powerful than the SQL's GROUIP-BY clause though it is harder to use. The GQL queries are

translated into QUEL queries for testing it with the INGRES system. This translation could also be used as the canonical form from which the query is translated to other languages. To this end the project [Webb88] being developed in the Computer Science department to translate QUEL queries to SQL queries could be used to translate QUEL output from GQL to SQL queries. The query in §2.3 is expressed in QUEL as

```
Range of D is Dept

Range of T is Item

Range if S is Store

Retrieve (D.name) Where

D.number = S.dept AND S.city=Christchurch AND

D.number = I.dept  AND I.price > 100.
```

2.3.3 QBE

QBE [Zlo77] technique is referred to as domain calculus as opposed to the original calculus and algebra languages. QBE uses a tabular technique as described in §1.3.3.2 and domain variables. Except for this the technique is same as the text based expression. The sample query of §2.3 expressed in QBE is shown in figure 2.1. By using table structure it has allowed for placing conditions in a more comprehensible manner. While the function tiles approach is as powerful as a purely text based expression the tabular technique is not sufficient to express all possible cases. QBE uses further techniques such as condition boxes and linking of rows by a common domain variable for ANDing instead of ORing of rows.

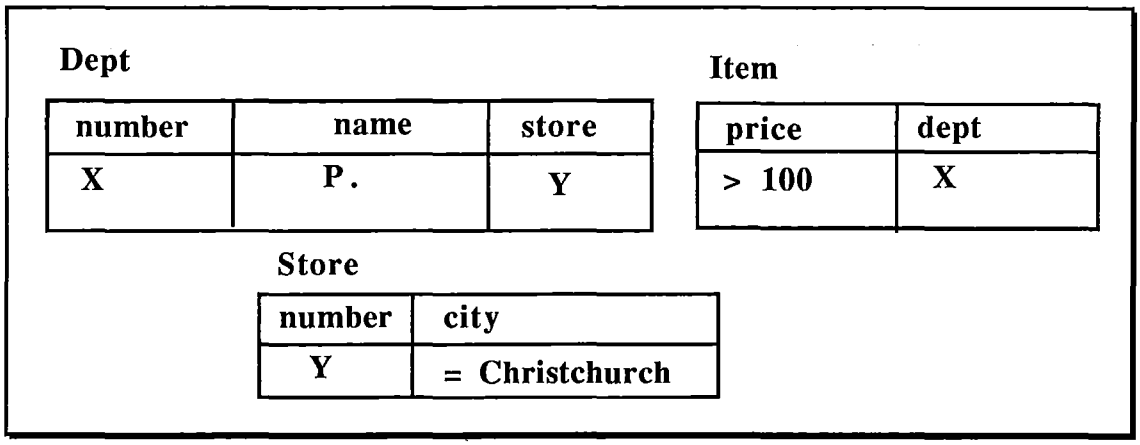


Fig 2.1

2.3.4 CUPID

The query of §2.3 expressed in CUPID [McD75] is shown in figure 2.2. The technique used in CUPID is same as the function tiles technique. Lines are drawn from operands of the function tiles to fields that are to be used. In cases where the operand is a constant a new object is created and line drawn to it. Thus as can be seen from the example, many lines have to be drawn to build an expression of even moderate complexity.

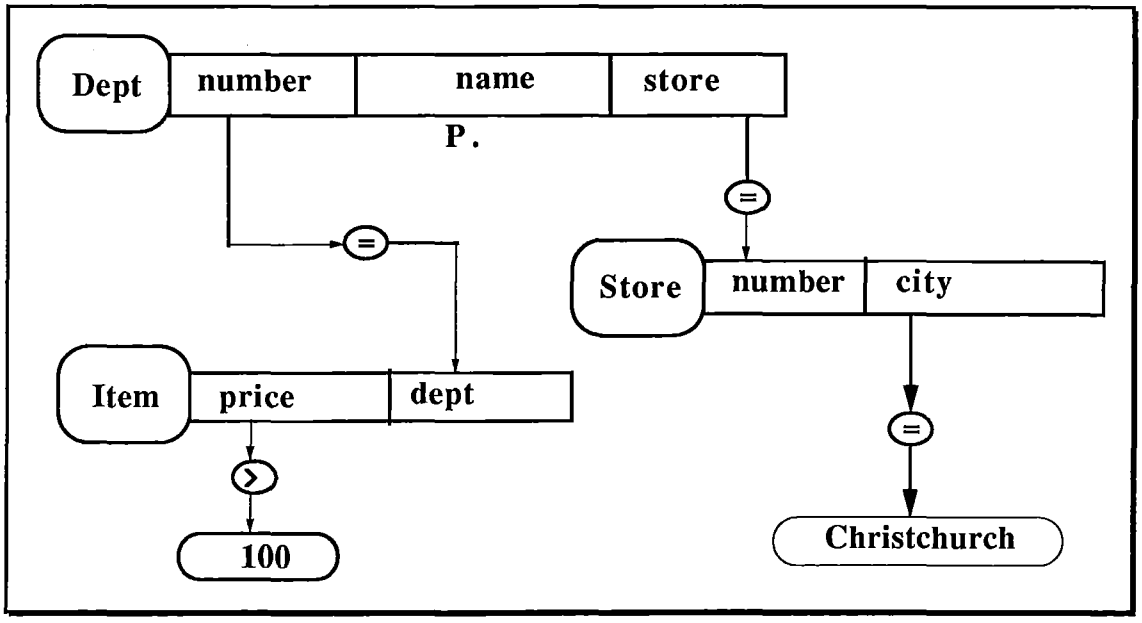


Fig 2.2

2.4 Discussion of GQL Techniques

QBE and CUPID are two graphical implementations that are distinguished from other graphical implementations discussed in chapter 4, because these are based on pure relational systems as defined in §2.2. Thus the techniques used by these two implementations are in fact graphical techniques for constructing expressions. Their similarities to techniques described in §1.3.3 can be noted.

The success of QBE suggests that a text based facility is still preferable provided the complexity of the expression can be diminished by breaking down the query into smaller units. The QBE approach is one step towards breaking down the query. It still requires the understanding of the concept of a variable. Both QBE and CUPID display all the details of a query in a single level. This in general increases the perceived complexity of the query by the user. Use of additional information layers as discussed in chapter 4 succeeds in breaking down the query components into even smaller comprehensible parts. In GQL, techniques discussed in §1.3.3.3 and the information layers described in chapter 4 are used to break down a query into modular units that can be be viewed at different levels of details. Thus GQL provides a tabular module (§5.5), subquery module (§5.10) and expression module (§5.6). The modular display of query in GQL has eliminated the need for scrolling to view query in most instances.

2.5 Results of Human Factor Studies and GQL

The quality of a relational query language can be measured by two of its attributes:

Selective power

User friendliness

Usually one has to be compromised to provide the other. Initial implementations discussed above have all been aiming to provide a relationally complete query language. Interfaces that provide the above two qualities in varying degrees will suit differing user communities. Human factor studies have been performed on some of the above

languages in an attempt to identify the difficulties users have in using them [Reis75], [Gree78]. In particular these studies concentrate on the users ability to understand the syntactic and semantic structures of a language and express queries using them. The results of these studies cannot be assumed to be universally correct for all of the user community. An easy to use query language, in addition to simple syntactic and semantic structures, should also provide the following features.

1. Information about the schema
2. Facility to formulate queries incrementally, using the results of previous queries.
3. Feedback to guide the user to formulate correct queries.
4. An easy medium to edit incorrect queries.

The above mentioned studies however do not measure these capabilities of the implementations. Since these studies are reported extensively and some the observations are intuitively correct, how they relate to the GQL implementation is discussed below.

Explicit use of AND and OR connectives were found to contradict their usage in English language. The tabular notation of QBE is found to be more suitable [Thom83]. GQL system uses tabular technique where appropriate.

Linking relation tables to define access path specification is found to be difficult [Reis75]. In chapter 4 various approaches to this problem is considered. The GQL system provides an information layer to assist users to formulate queries that require linking of relation tables.

Users have difficulty in expressing universal quantifier type queries. The way text based languages like SQL and QUEL are set up does not reflect the natural way people formulate queries [Thom83]. This is due to ambiguity in the English language and it is difficult to accommodate this ambiguity into a computer language. Natural language interfaces using expert systems is one approach towards accommodating this problem.

A graphical representation can diminish the difficulty faced by users to express this type of queries. In the current version of GQL simple aggregate values can be retrieved by simple selection (§5.5). GQL design permits some extensions to this (§5.12).

Reisner [Reis77] suggests a layered approach to query language design. The aim is to get users started with some easy to learn constructs and advance to more difficult ones in stages. This approach should eliminate having to learn too much to get started. In the GQL system the user can learn the constructs in the following stages. Each layer can be used without learning anything about the succeeding layers. More detailed examples are to be found in chapter 5.

1. Using predefined subqueries. To use GQL from this level all that a user must know is the name of the subquery that has been pre defined (§5.9).
2. Single relation queries. The list of entities can be perused to select the entity of interest and attributes from this entity as target list can be indicated by mouse selection (§5.5).
3. Join Queries where joins already exist. All relationships involving a selected entity can be perused and those of interest selected to formulate join queries by simple mouse selection (§5.8).
4. Qualified queries after learning to write qualifications. Attributes on which to place qualification can be individually selected (§5.5).
5. Writing expressions (§5.6)

The limitation of human capacity to process information has been documented [Mill57]. Thus enabling information to be modularised is an essential aspect of improving userfriendliness. The modular display emphasised in GQL as described in the previous section is in accordance with these results.

Chapter 3

Relational Query Language and Logic Expression

3.1 Introduction

A graphical formalism is developed in this chapter that has been useful in the design of GQL. The GQL technique itself is an extension of this formalism. The table concept introduced in this chapter is used in GQL. The table can be mapped into a tuple variable of a calculus based relational query language and provides the basis for translating a GQL query into a text based calculus language such as QUEL. The concept of a **connected query**, and aggregate queries based on connected query is explained. This is the set of queries that is proposed for implementation in GQL. Those queries that fall within the class of 'relationally complete' set of queries but are not considered for implementation in GQL are also identified.

3.2 A Graphical Formalism of Relational Queries

The graphical formalism introduced in this section is useful for explaining the requirements of a relational query language and identifying what can be expressed effectively using graphics. In figure 3.1 each of Table 1 to Table 4 represents a set of tuples derived from a relation table. Q1 to Q4 represent simple qualifications placed on the fields of the Table rectangles. A simple qualification is one that does not involve logical AND or OR operators but only involves scalar comparisons of two alphanumeric values. In general a simple qualification may involve any number of fields from any number of Table rectangles and yields the result true or false. The middle rectangle highlights the Table rectangles involved in a given qualification by indicating a black dot in the appropriate position. In figure 3.1 the qualification Q1 involves fields from Table 1 and Table 3, the qualification Q2 involves fields from

Table 2 and Table 4 and so on. In figure 3.2 the example query used in chapter 2 is expressed using the above notation.

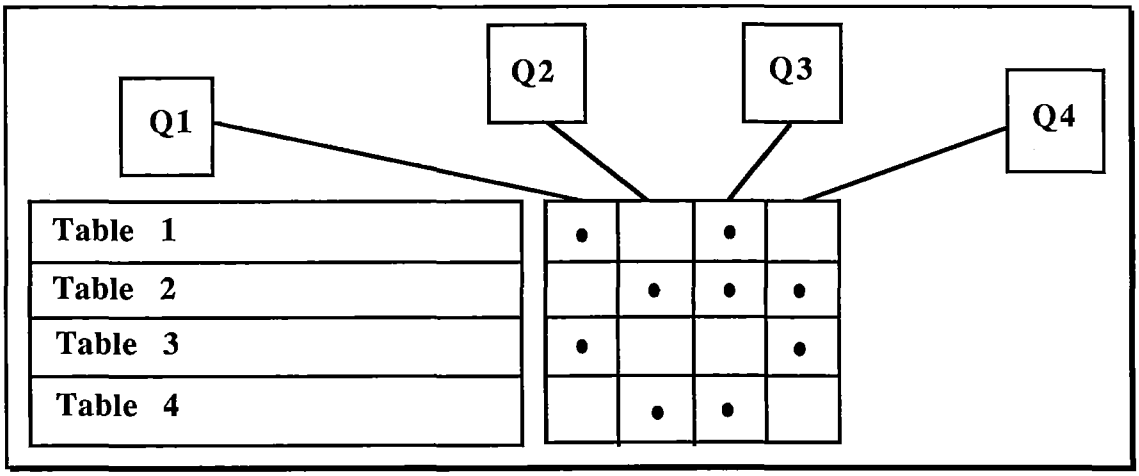


Fig 3.1

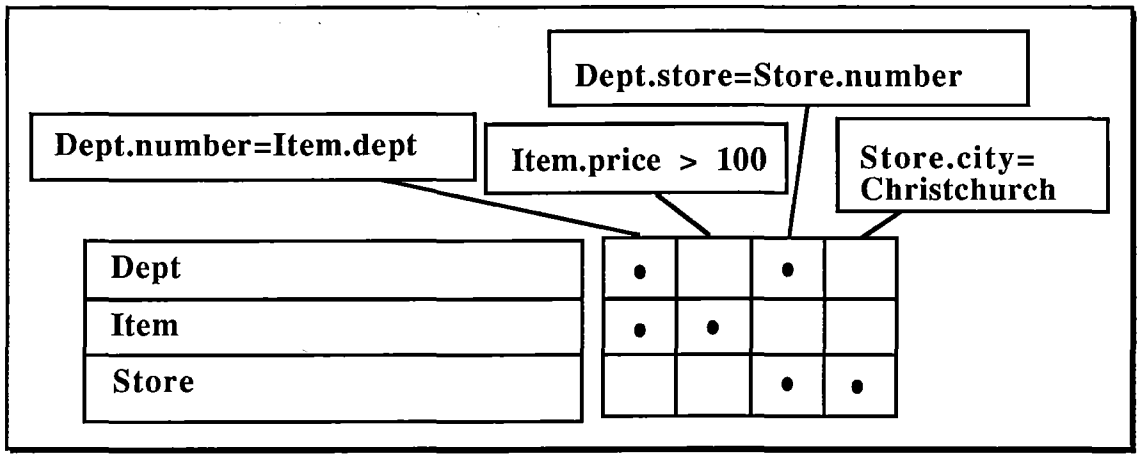


Fig 3.2

3.3 Query Classification

The following classification of relational queries has been proposed [Jark85] as an increasing set that is useful in identifying classes for providing special constructs to express them.

Single relation queries
Universal relation queries
Conjuctive queries
Disjunctive queries
Universal quantifier queries
Horn clause queries

In the following sections a classification suitable for graphical representation in GQL is developed and figure 3.3 shows how this classification compares with the above.

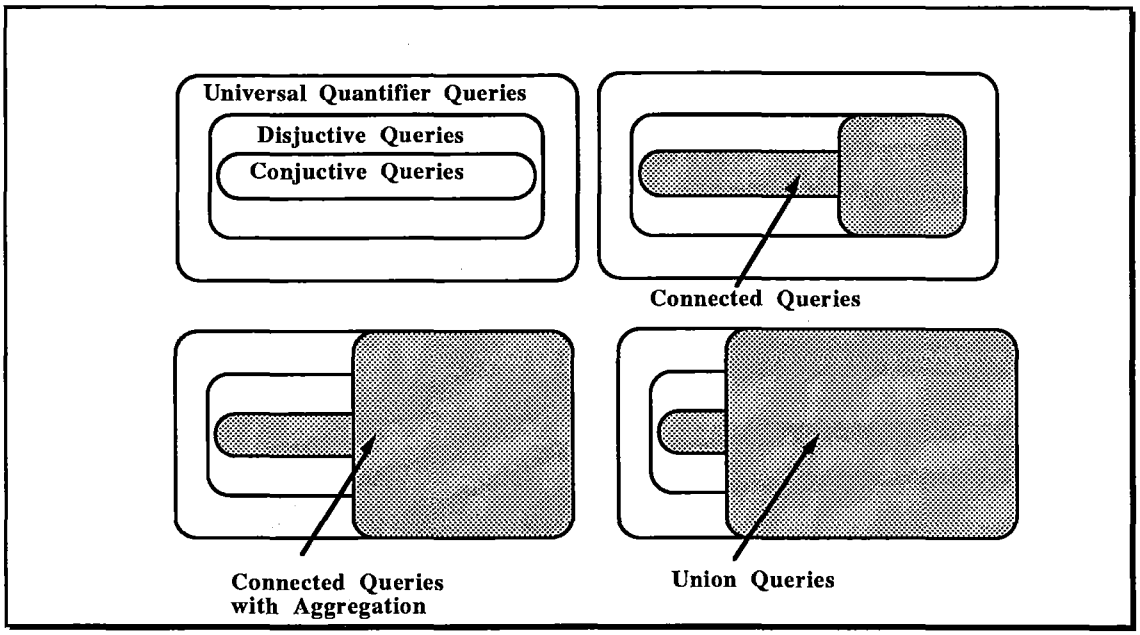


Fig 3.3

3.4 Connected Queries

In the formalism described above if all qualifications in the qualification rectangles are implicitly ANDed then the set of queries represented by this form of representation is the set of conjunctive queries as specified by the first classification. Under such a qualification the Table rectangles will each represent a set of tuples that satisfies the qualifications. Thus it is possible to produce a cartesian product of all these Table rectangles resulting in a large table, where every tuple in this large table satisfies the qualifications.

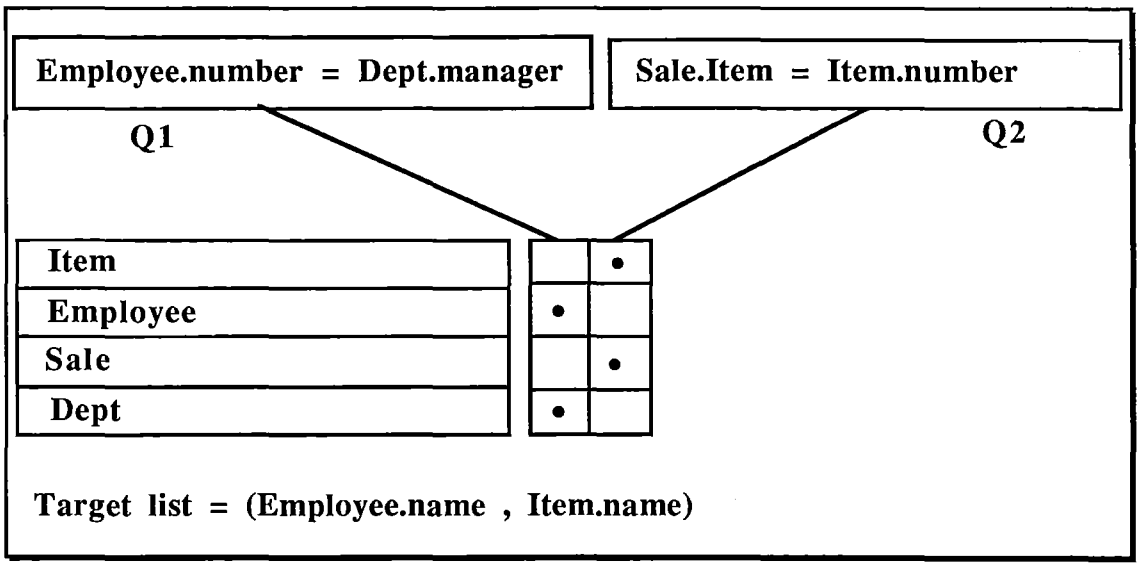


Fig 3.4

The concept of a connected query is defined for use in GQL. A query is defined as **connected** if it satisfies the following conditions.

1. A qualification is said to connect the tables from which it derives its operands.
2. All the tables in the query are connected together by a set of qualifications which are logically ANDed.

By this definition the query shown in figure 3.4 is not connected. This query will retrieve all possible combinations of **Item.name** that satisfies qualification Q1 and **Employee.name** that satisfies qualification Q2. Although in a database environment the ability to satisfy such queries must be available such a query can also be used mistakenly by a casual user. In most instances data meaningful to an end user cannot be retrieved by a query where the tables are not connected according to the definition above. A friendly user interface should disallow a user from writing a meaningless query whenever it is feasible to do so. This is especially so because most such meaningless queries can result in the retrieval of very large amount of data.

Thus testing for connectivity ensures, cartesian product of two sets of tuples are not produced as result, where the two sets of tuples represent independent units of result. GQL performs this connectivity test before using a query to retrieve data from the host

system. The connectivity is also used in GQL to select a connected query for conversion to a subquery (§5.9). The algorithm used for this test is described in §7.4.

The connected query as defined above is well suited for tabular techniques of §1.3.3.2. Each table of the formalism can be expanded to display all its fields where additional conditions can be placed using tabular techniques. Thus some disjunctive queries, where the disjunction of conditions is limited to those that can be expressed within a table, are included in the set of connected queries as shown in figure 3.3. Another approach could be to apply the tabular technique to a single table formed by combining all the tables in the connected query. GQL uses the first approach.

Other instances of valid queries that will retrieve meaningless result are possible. These are due to semantic misinterpretation of the schema [Howe83] and have been given the name 'connection traps'. In a graphical interface like GQL the possibility of such misinterpretations can be reduced by graphical representation and by careful naming of objects (§4.5.4).

3.5 Union Queries

The connected query does not include all the possible valid queries. The union queries are aimed at expressing this set of queries. Consider the query in figure 3.5. According to the definition in the preceding section this is not a connected query because the three tables are not connected by a set of logically ANDed qualifications. However if the target list is specified as **(Item.name, Item.price),** where all fields are from the table **item** then the retrieved result is a meaningful unit. If the target list is specified as **(Item.name, Dept.name)**, then the retrieved result is not meaningful. This is because **Item.name** that do not satisfy qualification Q1 are included in the result due to ORing of Q1 with Q2. This type of query is therefore only valid for a restricted set of target list. To enforce this restriction the UNION operation of the Algebraic formalism is proposed, thus incorporating some procedurality into the

language. Figure 3.6 illustrates this. The 'u' in the middle rectangle indicates the tables to be unioned and 'r' in the middle rectangle indicates the result table of the union.

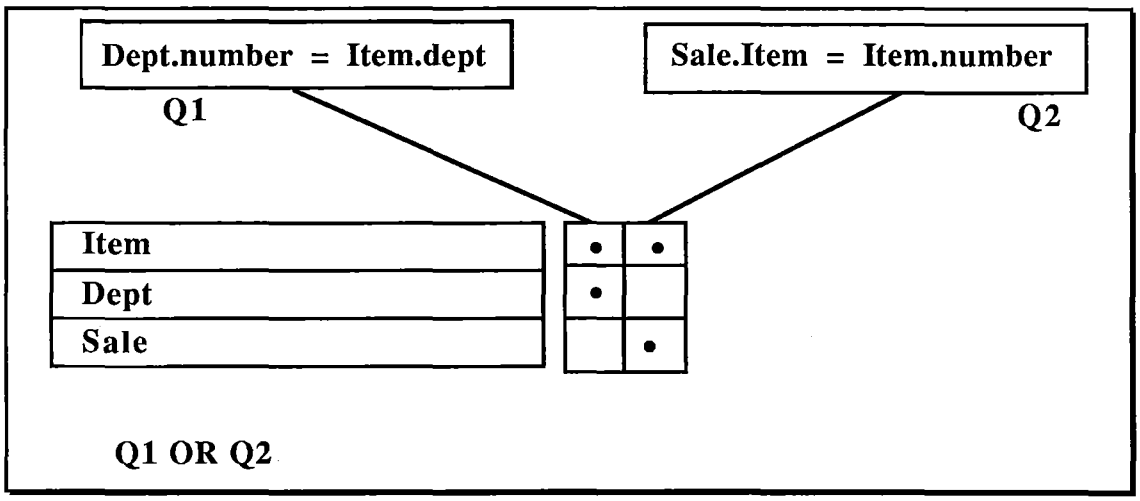


Fig 3.5

A **union** query is formed using a set of compatible tables. As can be seen the **union** query may involve many connected queries. The restriction is placed that the target fields should come from a single connected query within the union query. The result table formed by the union must be a component of this connected query containing the target list. The need for such elaborate specification to maintain integrity arises due to the degree of freedom given to the user in expressing the query. A query definition process should therefore use the definition of connectivity to ensure that these constraints are imposed.

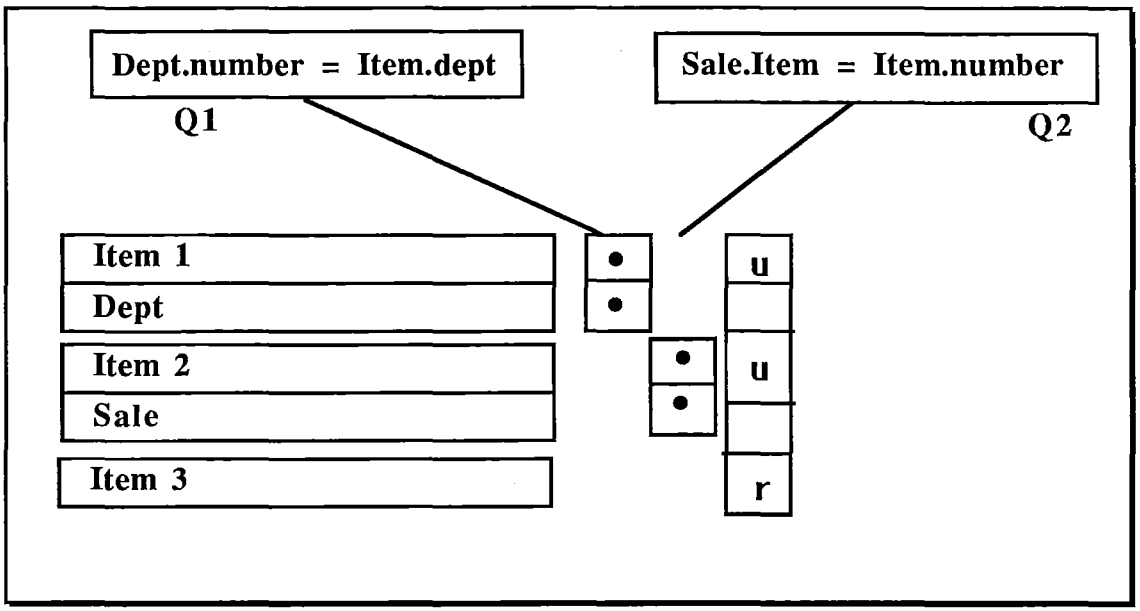


Fig 3.6

The algebra Intersection operation is not necessary, since the above constructs are sufficient to express all queries that can be expressed with the addition of Intersection operation. Due to the complexity they will add to GQL implementation, union queries have been omitted.

3.6 Aggregation

Most commonly used queries require the formation of a hierarchical view of some information stored in the database. These views in general need not be based on the base relations in a relational database. The HIQUEL language (§4.3.3) provides an interface for forming such views based on the implemented relations. Such a view may also require aggregate values of fields belonging to an object at the next level down. Ideally a user interface should provide facilities for forming such views. The GQL interface is suited for graphically displaying this hierarchy (§5.12).

The relational query languages based on the relational algebra and relational calculus proposals are tuple oriented and do not in themselves have the ability to form such hierarchical views. Constructs provided in traditional query languages allow for

grouping tuples over some domains in the tuples and evaluating the aggregate values over these grouping for retrieval or placing a condition on these aggregate values.

Thus a hierarchical query view as in HIQUEL can be built where only the highest level owner and its aggregate properties can be retrieved but aggregate conditions can be placed on the aggregate values generated on the lower levels of the hierarchy. This can then be translated into a traditional query language. The current implementation of GQL only allows for retrieving set of simple aggregate values over fields of tables. This can be extended to include the aggregation features as described below.

3.6.1 The Requirements of Aggregation

The following requirements are identified for expressing this type of query :

1. Aggregation is meaningful only within a connected query.
2. Generation of tuple sets. The first step in specifying such queries is to construct the set of tuples over which qualifications are applied. This can be divided into specifying constant sets and set variables.
    (a) Constant set: The set of tuples represented by a Table rectangle in the above formalism is a constant set.
    (b) Set variable: The set obtained by grouping tuples over some domains is a set variable that represents different sets for different domain values over which they are grouped.
2. Requesting aggregate values: Aggregate values can be requested over a constant set or a variable set.
3. Placing set conditions on a variable tuple set. This can be
    (a) A scalar condition on aggregate value
    (b) Set comparison with other sets

3.6.2 A Method for Specifying Aggregate Conditions

Specifying the above requirement in the formalism used in this chapter can be achieved as follows.

1. Generation of variable tuple sets is specified by indicating a set of owner tables and a set of member tables through an aggregation operation. The tuples of the member tables are grouped over the tuples of the owner tables having common values. In this method arbitrary grouping over any set of attributes is not possible. This is not a disadvantage in an end user interface since using such features has been found to be difficult for the end users [Thom83] and most commonly used aggregation can be achieved by grouping over the key fields of a relation in the database. Grouping over a whole table is equivalent to grouping over key fields of the table.
2. A grouping specified above will result in the formation of an aggregate table whose attributes are aggregate functions applied over an attribute of a child table. Using this technique attributes of the aggregate table can be retrieved in the target list. Scalar conditions can be placed on these attributes of the aggregate table using tabular technique, thus achieving the requirement 3a of the previous section.
3. Treating table rectangles as constant sets and aggregate tables as variable sets comparisons can be made among these sets satisfying the requirement 3b in the previous section.

Figure 3.7 illustrates an aggregate operation formed over **Sale** as the member denoted by **m** in the middle rectangle and **Item** and **Dept** as the combined owner tables denoted by **o** in the middle rectangle. Thus the resulting aggregate table denoted by **r** (for result) in the middle rectangle can have an attribute **Sum(Sale.quantity).** Retrieving this field with **Dept.name** and **Item.name** is equivalent to retrieving the total quantity of each item sold by each department. Placing the condition '>10' on the attribute

Sum(Sale.quantity) will be equivalent to retrieving departments and Items whose sale through that department is greater than ten.

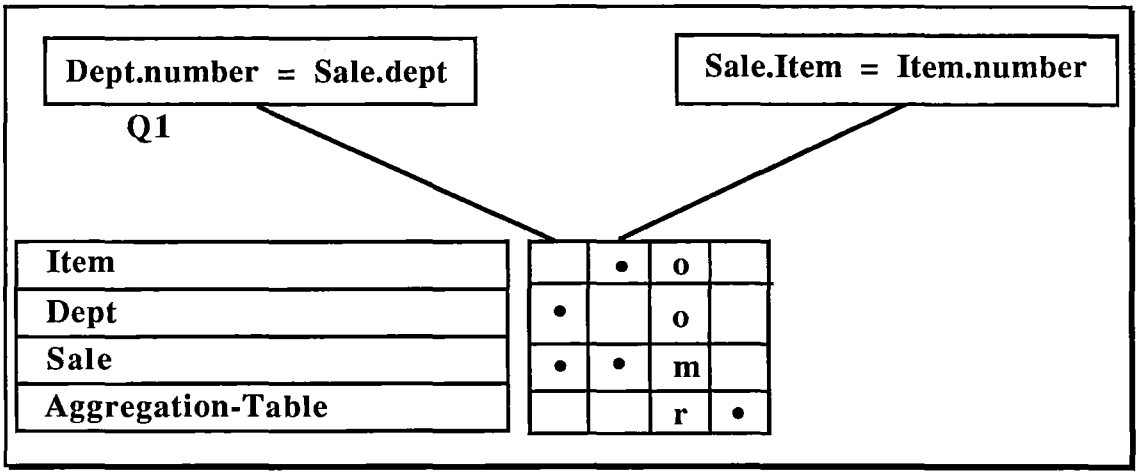


Fig 3.7

3.6.3 Expression of Aggregation Operation in QUEL and SQL

In the query in figure 3.7 if the target fields include an attribute from the **Sale** table such as **Sale.date** and an attribute from the aggregate table such as **Sum(Sale.quantity)** then the result tuple will not form a meaningful unit. The QUEL language however permits such retrieval thus producing a cartesian product of **Sale.date** and the aggregate values. In SQL the specification of such a target list is invalid, since all target fields other than aggregate values must be specified in the GRP-BY Clauses. Therefore in GQL it is necessary to restrict the target fields to highest level owner of the hierarchy and its aggregate properties.

Translation of grouping required by the aggregate operation can be achieved using the 'Group-By' clause of SQL or the BY clause of QUEL over the key fields of the table concerned. If key fields are not known then the grouping may be specified over all the attributes of the table. However this will result in an untidy translation. In QUEL translation if the grouping is specified over key fields the resulting query will be still valid, however in SQL all the target fields in the Select clause must be specified in the Group-By clause. Thus grouping over all the attributes of the table may be necessary.

In addition SQL queries containing more than one Group-By clause must be nested. The following conclusions can be drawn from the considerations of translation.

1. Key information of relations in the schema must be captured
2. SQL translation requires fields in the select clause to be included in the Group-By clause. Thus a hierarchy structure must be defined over a connected query where a table can be in only one Group-By clauses as member, while it may be in a number of Group-By clause as owner.
3. An alternative to grouping over key fields is to specify the grouping attributes as in SQL or QUEL, using the Attr-Grp structure as defined in GQL local dictionary (§6.5).

More discussion on aggregation in regard to applying the above concepts to GQL technique is found in §5.12.

3.7 Negation Handling.

A shortcoming of relational query languages formalism is the lack of its ability to express negated queries naturally. A query such as 'Supplier who does not supply P2' is a negated existential query [Gray84]. But the query language formalism does not permit negation of this form. The query thus has to be expressed as a query involving aggregation where the set of parts supplied by a supplier does not include P2. A direct negation of a connected query is more difficult to interpret in standard query language and is not considered in the GQL system. The use of the PROLOG language as an interface to relational databases is one approach that is attempted [Deyi84] to provide an interface for expressing such logic queries.

Chapter 4

Semantic Modelling and Query Language

4.1 Introduction

The field of semantic modelling [Hull87] deals with capturing and representing information in a structured way. The ability of a user interface to provide an application independent front end that responds intelligently to users requests is very much dependent on the ability to capture semantics within a structure. Semantic modelling efforts are also aimed at other database issues such as schema design aids, integrity maintenance and smooth evolution of the database. In this chapter some semantic modelling approaches and their contribution to improved user interfaces are discussed and some directions towards the design of GQL are noted.

4.2 Automatic Access Path Determination Model

The theory of normalisation [Date86], [Ull82] reduces the real world data into tables with specified properties which can be shown to be desirable. Thus in a relational data model information is stored in tables generated through normalisation. An end user perceives data of interest to him as tables which in general need not be the same as the tables produced through normalization. Thus the user is left with the responsibility of defining the table that he requires as well as specifying conditions the result table must satisfy.

The access path determination model as a user interface was first proposed in [Kent81]. The aim here is to define the domains of the database distinctly and name them uniquely. When the user requests certain domains as of interest the domain structure can be used to determine the equi-joins required to generate the table perceived by the user. In general a unique path deduction by this method is not guaranteed. Figure 4.1

shows the domains and relations of part of the sample schema. For a query that requests **Item.name** and **Employee.name,** there are three possible access paths that can be deduced. They are,

1. Item → DeptNo → Dept → empNo → Employee
2. Item → ItemNo → Sale → empNo → Employee
3. Item → DeptNo → Dept → empNo→ Employee → empNo → Employee

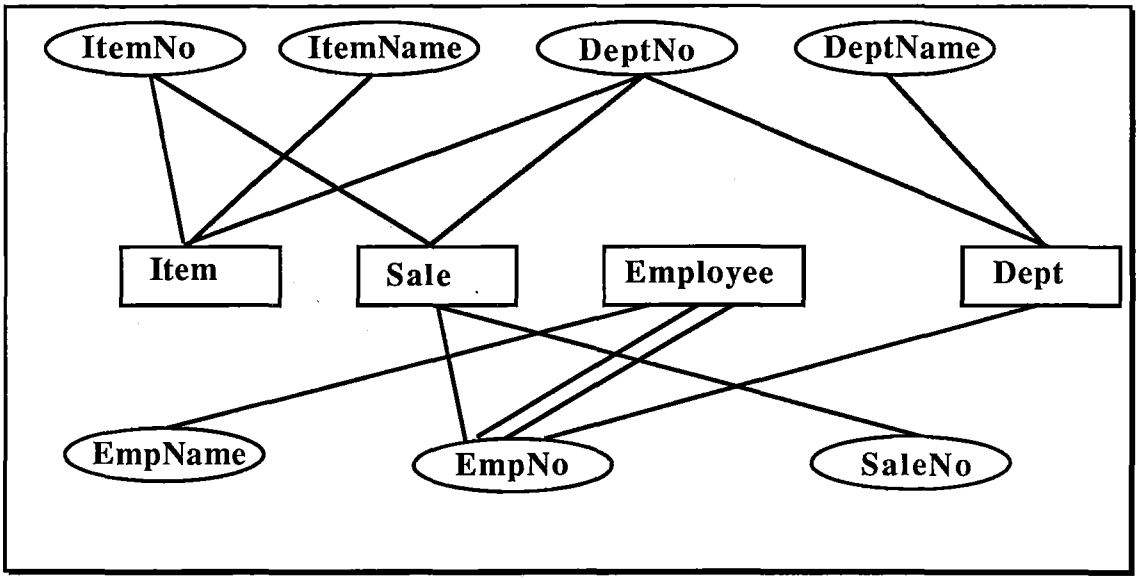


Fig 4.1

Methods for resolving these ambiguities have been proposed. The transaction processing system [Mar80] is based on one such approach where the domain information and primary key field information of relation tables are used in an algorithm to derive an access path. The advantage of this approach is that by automatically deducing the access path the system can permit the user to express the queries using the tabular technique which has been found to be easy to use (§2.5). Such an approach can be made to work by designing the database to limit the number of possible ambiguities as those listed above. The access path through a relational data model represents relationship that exists in real world between entities.

Developments in semantic modelling techniques that represent this fact explicitly in the model are discussed in the following sections. These have proved to be more useful in determining access paths compared to the universal relation [Kent81] approach. If the access path required by a query is defined using other techniques the idea of using a table to express the remaining conditions of the query is still a feasible idea. The INGRES Query By Form tool uses this idea. The user can express queries using a form where the required join of two relations can be predefined. This approach was considered as a possible alternative for the GQL system. The user can define the access path for a query using an interface currently available in the GQL system (§5.7). But additional conditions can then be applied at a separate level either by using tabular techniques or function tiles techniques as in §1.3.3. The disadvantage is that a two level definition of a query can remove the comprehensibility of a query.

4.3 Entity Relationship Model

The entity relationship model, with its associated diagramming technique, was first proposed in [Chen76]. The proposal stayed close to the pure relational database model. As a result it has been the basis for graphical interfaces to relational systems in the database access and data design area. Many extensions to the original notation have been adopted [Mart85a], [Teor86]. The contribution of this modelling approach and its diagrammatic convention to query languages are studied in this section. Figure 4.2 is the entity relationship diagram for the sample schema in appendix A. The entity relationship schema should carry additional information about the database that is not carried in the pure relational schema.

The most obvious use of such a diagram in a query language is in defining the access path. If the user is presented with such a diagram selecting the required entities and relationships is a simple task. This is in contrast to the linking that must be performed even in QBE the most userfriendly of the languages surveyed in chapter 2. This approach is also preferred to the access path deduction procedures of the universal

relation model since it does not carry the ambiguities of the universal relation approach. Three implementations that make use of the entity relationship model as a query interface are studied below. The three implementations make use of the entity relationship model in three distinct ways, illustrating the three areas of usefulness of this model as a query interface.

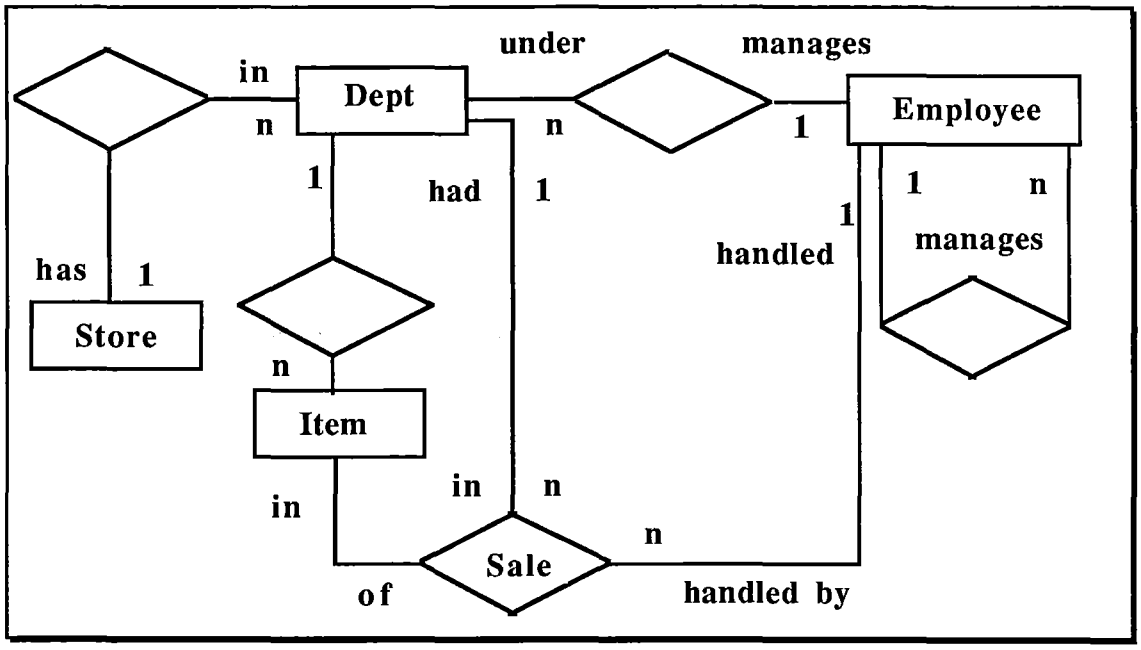


Fig 4.2

4.3.1 ER Diagram as Query Interface

[Zhi83] uses such a technique. In this approach an entity relationship diagram is presented to the user and the user can select the entities and relationships from this diagram. If the user selects immediate neighbours that are directly connected by a relationship then this step will uniquely define the access path of a query. However if the selected entities and relationships are not immediate neighbours then the access path must be deduced by the system. If the path cannot be deduced uniquely due to cyclic access paths [Zhi83] in the schema then problems similar to the universal relation approach will have to be resolved. Since the entity relationship diagram is in front of

the user requiring the user to define the path fully is an acceptable solution. Additional conditions for the query are then placed using separate tables for each entity that is selected.

4.3.2 Suitability of ER Diagram as User Interface

In the techniques described in the previous section once the user has selected the access path retaining the rest of the schema is irrelevant. Thus this can be deleted from the screen. [Lar87] suggest such a technique where by the selection step is indirectly performed by deleting the irrelevant parts of the schema. If the flexibility of adding entities or relationships to the query at any stage is to be maintained then an additional facility is required. Through this the user should be able to add access paths to an existing query. This can be achieved by displaying objects of the schema neighbouring the current objects in the query. As discussed in [Fogg84] algorithms for performing this task themselves can be very complex, since this requires intelligent placing of the box and line diagram. In GQL the user can develop a query starting from any relation in the schema. Access paths are selected from a list of immediate neighbours to any of the relations in the query or by selecting a relationship between two of the relations in a query. Thus the above placement problem is eliminated in the area of access path definition.

GQL divides the query definition area into left and right regions for placing the entities and uses the middle region for the relationship display (§5.2). The difficulty in drawing an entity relationship type diagram without human aid is avoided by this technique. This technique also permits the subquery usage in a query to be constrained to the middle region, thus designating this region for access path definition and subquery constraints. This separation results in an easy to comprehend display as well as an easy to implement interface.

Another shortcoming of the use of entity relationship diagram as a query interface arises when a query itself requires a cyclic access path definition. When two sets of tuples from the same entity are required in a query, the entity must be displayed twice. The example shown in figure 4.3 is a GQL display of a query from the sample schema that involves a cyclic access path. In an ER diagram interface this can be expressed clearly. However consider the query '*Get departments managed by employee who handled the sale from hardware department on a particular date*'. The access path required by this query in a GQL display is shown in figure 4.4. Here two sets of tuples from the **Dept** entity are required. To express this query using an ER diagram is not possible. Some modification to ER diagram is required, thus adding more complexity to the interface implementation.

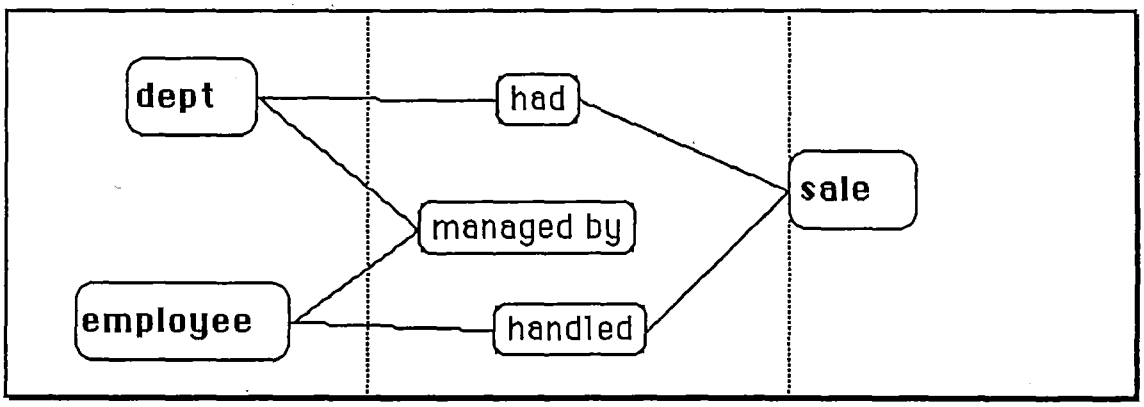


Fig 4.3

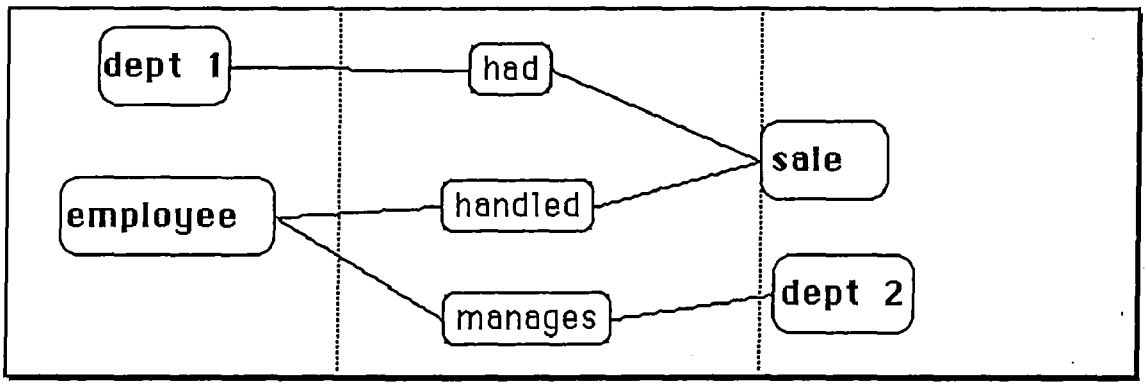


Fig 4.4

A strict entity relationship diagram of the schema also eliminates the option of using predefined views in the host system as the starting point for query formulation. From the foregoing discussion the following conclusions can be made about the use of entity relationship diagrams for querying.

1. Since the aim of the project is to develop an interface to a relational schema the information required to present the Entity Relationship diagram must be first built into the database before such an approach can be taken.
2. An intelligent drawing routine that places entities correctly is required. In such an approach the entities and relationships not required in a query are deleted from the ER diagram first. Thus if the user wishes to add an entity or relationship to a query that has already been developed then the application should allow the user to do this. If neighbouring objects are thus to be displayed for selection it is essential that good placement algorithms for objects are provided.
3. Cyclic queries involving the same entity twice in a query are not clearly expressed through ER diagrams.
4. Views defined in the host system cannot be used for query expression through such an interface.
5. Presenting the whole schema to the user is not desirable, since this adds to the perceived complexity of the interface.

4.3.3 HIQUEL

The HIQUEL [Urs83] language exploits another aspect of entity relationship modelling. The cardinality of the relationships is recorded in the schema. Using this information a hierarchical view of a subschema can be formed. If the user indicates an object as the root of the hierarchy then the cardinality of the relationships emanating from this object is used to determine if the neighbouring objects in the access path are its children or siblings. Thus if the cardinality of the relationship from the root to a

neighbouring object is 1:N then the neighbouring object becomes a child otherwise if it is 1:1 then it becomes a sibling. Applying this to succeeding objects in the access path the hierarchy can be defined. Using this hierarchical structure aggregate conditions can be placed easily.

Aggregate queries are the most difficult queries to express in the traditional query languages surveyed in chapter 2. The HIQUEL technique provides an easy way to express a subset of this kind of queries. The grouping permissible with the HIQUEL facility is limited to entities and their immediate relationship objects only. The approach is useful if the underlying schema is built over a strict entity relationship model. Then most of the aggregate queries required by the end user can be formulated using entities and their associations as the units of aggregation. The aggregation formalism described in chapter 3 is similar to HIQUEL technique but permits arbitrary grouping of tables. Compromising power for userfriendliness has resulted in an easy to use aggregation feature in HIQUEL language.

4.3.4 LID

The LID [Fogg84] implementation uses the entity relationship diagraming technique to provide a browsing interface. The neighbourhood objects that have at least one link with the current tuple in the current entity are the only ones displayed. This gives a very visual feedback while browsing. The current entity and the current tuple can be changed by the user thus browsing through the database. The difficulty encountered in implementing such an interface is the drawing algorithms for dynamically placing the objects.

4.4 RM/T Model

No implementation known to the author exists that is based on this model [Codd79]. Some of the proposals of the RM/T model that are directly relevant to this project are

discussed in this section. The following correspondence can be noted between the objects in the RM/T model and the entity relationship model.

RM/T	**ER**
Characteristic Entity	Weak Entity
Kernel Entity	Entity
non Entity Association	Relationship
Associative Entity	.....

The RM/T model proposes a set of system relations to record the access path definition among the above objects. The n-ary relationship of the ER model is recorded in the RM/T model as **n** binary associations between the associative entity and its participating entities. The definition of associative entity facilitates the definition of associations on associative entities. If the distinction between associative, characteristic and kernel types is removed then the RM/T model records the access path as a set of binary relationships among the relation tables. The GQL local dictionary structure described in chapter 6 in fact resembles such a system. Another addition to this model is its facility to define subtypes for any type of entity.

The distinction between kernel, associative and characteristic entities in a global schema definition is useful as a design aid and for integrity maintenance. A casual user querying a large database is only interested in part of this global schema. Thus in an end user interface this distinction does not serve any purpose and may in fact cause unnecessary confusion if these types are displayed differently. In addition, since GQL is to interface to any relational schema, this information may not be directly applicable to the relations in a relational schema not designed strictly according to this semantic model. A relational schema that is designed without strictly adhering to a particular model may compromise some of the semantic distinctions for the sake of performance efficiency.

In the sample schema the **Item - Dept** relationship which is in fact a many:many, is recorded without any associative entity since in most cases the relationship will be one to many. Occasionally an item may be sold from another department in a store where the appropriate department does not exist. Such compromises are always made in schema design, therefore applying the above distinctions for any relational schema is not appropriate. The simplicity of a single type is therefore preferable in the GQL system. For the same reason maintaining update integrity in a standard manner across all database schemata is not easy to achieve. The subtype category of RM/T model too cannot be applied to relational schemas in general. But GQL's subquery layer can be used effectively to apply subtyping to any relation in the schema.

4.5 The GQL Technique : A Preview

In addition to the desirable features for a query language listed in §2.5, the following objectives are aimed for in the design of GQL:

1. A graphical language that could be used as a front end to any relational database schema independent of a particular semantic model.
2. An ability to build in information layers about the model that are useful from a query language point of view.
3. Ability to view levels of details through simple user actions.
4. To preserve reusable parts of query and the ability to use these in building new queries.
5. Ensure correctness of a query by presenting only valid choices to the user while building it.

GQL maintains a local dictionary to achieve some of the above objectives. In chapter 6 three layers are identified in the GQL dictionary. Of this the first layer is the minimum amount of information about the schema that must be extracted from the host system. The second layer is used to represent the semantic information as discussed in this chapter. The third consists of predefined queries. The emphasis on the information

layers of the GQL system has been its usefulness in query building. GQL provides a uniform interface for defining queries and relationships. Additional relationships can be added to the GQL dictionary at any stage.

4.5.1 Associations in the GQL Dictionary

The access path information as a set of binary equi-joins, one per relationship, between relations of the schema is one layer that is identified as useful in GQL. As in entity relationship diagrams these associations can be depicted graphically for good comprehension. These relationships are named to inform users for selecting access paths for a query. All types of associations identified in the RM/T model and entity relationship model can be expressed as binary equi-joins in an interface that is to facilitate easy access path definition. The description about GQL in this and the following chapters refers to the term relationship to mean a binary equi-join between two schema relations. The term relationship in real world could mean an association that involves many objects. The binary equi-join will represent the whole of such a relationship if only two objects are involved in it. If more than two objects are involved then a binary equi-join represent only a part of that relationship. The rest of the information required to fully describe the relationship will include additional relations and equi-joins. One of the additional relation will be an association entity. However for better readability the term relationship is sometimes used in preference to equi-join.

The implementation details of associations in the GQL local dictionary are described in §6.6. These can be added to the dictionary through an interface described in §5.6. If GQL is implemented as part of an integrated interface which includes data modelling tools then it will be feasible to extract some of these associations from this level as well as add additional associations through the GQL interface.

4.5.2 Cardinality of Association

The cardinality of the binary association can be recorded in the GQL dictionary and used in formulating aggregate queries as in HIQUEL (§4.3.3). Every equi-join used in a query can be turned into an aggregate operation resulting in an aggregate table, by user action. The cardinality can be used as in HIQUEL to determine the owner member information of the association. If the association is many to many then the user must specify the owner member detail required in the query. Another approach could be to allow the user to specify the owner member detail of the aggregation operation in all instances. This results in greater power and less userfriendliness. The aggregate formalism of chapter 3 is thus reduced to aggregation involving two tables only. It must be emphasised that the usefulness of this technique relies on the underlying schema being defined according to the semantic model. In the sample schema a tuple in the **Item** relation does not strictly represent an entity item, but the concept of an item sold from one particular department from one particular store.

4.5.3 Membership Class of Association

Membership class of an association cannot be directly used in query formulation. Its main usefulness is in maintaining integrity during updates. Thus the deletion of a tuple may be disallowed if it owns a mandatory member in an association. Alternatively a deletion may cascade to deletion of all mandatory tuples belonging to this association. The update ability has not been added to GQL. The issue of database updates is closely linked with the integrity maintenance of the database. Integrity considerations usually involve restrictions to database updates and cascading the effect of the update. A GQL user tends to deal with subschemas, therefore permitting updates that effects the full schema is not desirable.

The basic integrity rules for a relational database are stated in §2.2 as the primary key integrity and referential integrity. Most databases will involve the enforcement of

additional integrity rules. The method of enforcement applied may vary among relational implementations. In INGRES integrity rules may be specified as QUEL statements. Many of the application specific integrity rules are enforced by restricting the updates through specialized update routines. Thus handling of these issues varies among the relational database systems.

4.5.4 Graphical Representation of Association

Connections that represent a relationship between entities can be shown graphically. Connecting text can be added to this display that describes the real world relationships between the entities. This can give helpful clues when the user is building an access path using these joins. Thus a GQL user is able to develop the access path by incrementally selecting the equi-joins that are required from a list that is presented. In §5.8 this process is illustrated. With this technique the shortcomings in using entity relationship diagram for querying as noted in §4.5 is eliminated.

Misinterpretations of the access path through associations by casual users have been noted. This problem is related to the 'connection-traps' discussed in [Howe83]. Figure 4.5 shows a schema for customer supplier parts database. The supplier - part connection represents the suppliers who supply a particular part. The customer - part connection represents the customers who buy a particular part. If the supplier - customer pair is retrieved the user can assume that it represents the suppliers who supply the customers, which is not the case in this schema. In a graphical representation of associations it is easier to recognise such connection traps.

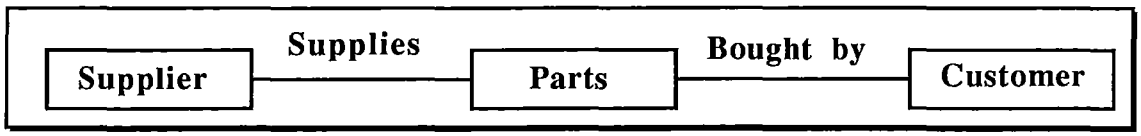


Fig 4.5

Chapter 5

Graphical Techniques in GQL

5.1 Introduction

The notion of ease of use is something that pervades the whole process of design. The design considerations described so far in chapters 3 and 4 lead to some design decisions. In addition to these design considerations a good user interface is produced by attention to details. The importance of these are extensively reported in human computer interaction research publications [Gai83].

A parallel can be drawn between the design of a computer interface and the design of a home appliance. One aspect of the design of a home appliance unit is the use of high technology to provide sophisticated features such as automatic turn on and off etc. The other aspect of the design is to develop the interface between the user and the appliance. A poor interface design is illustrated in placing the four switches of a cooking stove elements in a line while the elements themselves are placed on a rectangular top. A rectangular placement of the switches will provide a much easier to follow visual clues. Given the Macintosh user interface the design problem to be addressed in the GQL system at this stage is parallel to this interface design. This chapter describes the GQL graphical interface and the reasons for some of the techniques employed.

5.2 The Query Display Window

A graphical interface to relational databases includes the query display and the result display. The distinction and interconnections between them may vary for different implementations. In a browsing environment such as LID described in §4.3.4 the two cannot be distinguished whereas in a static retrieval type of environment they are more

or less independent. The emphasis of this project has been on the query formulation with a display window provided where results can be viewed.

The concept of subqueries is used throughout the GQL system. Since an essential use of this concept is in the modular definition of queries the use of separate windows for each subquery is not appropriate. Allowing the user to open an unlimited number of windows one for each subquery can result in the user getting disoriented. Thus only one query display window is available in the GQL system and techniques for viewing subqueries through this window are discussed in §5.11. The query display window is divided into four rectangular areas as in figure 5.1 and the reasons for this are described below.

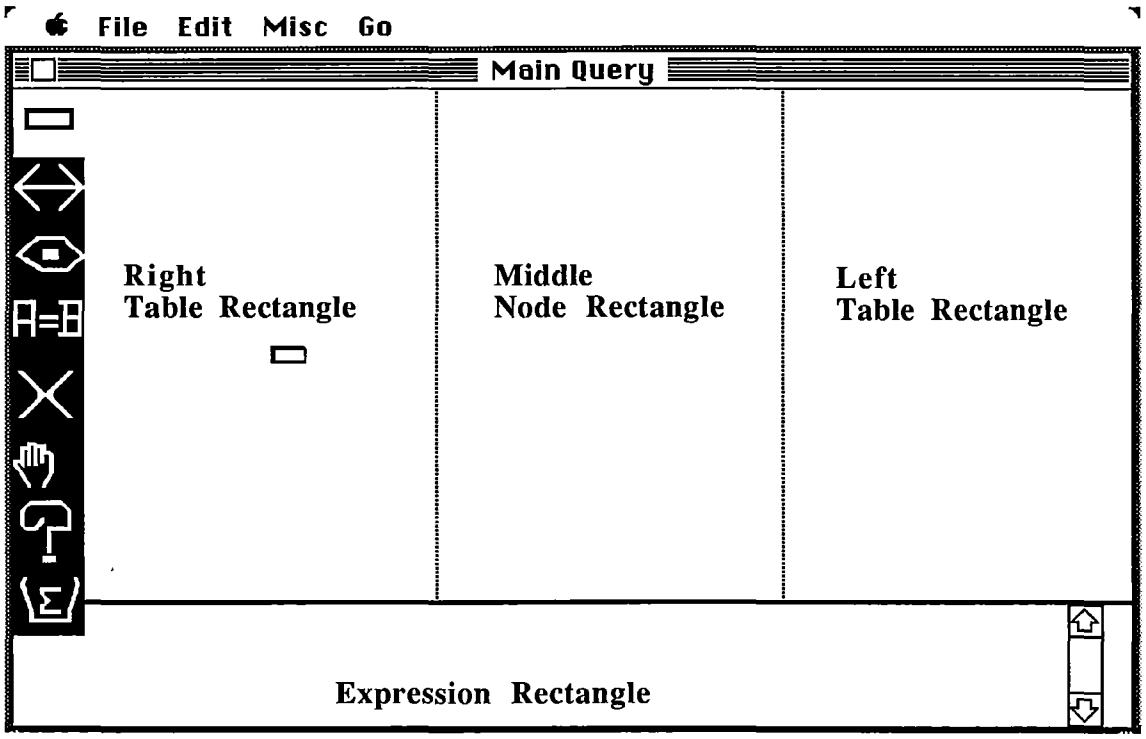


Fig 5.1

5.2.1 Palette Rectangle

By using tools selected from the palette to specify actions the mode change in the interface can be kept to a minimum. Since all queries are viewed through a single window and all operations using tools are directed to this window this rectangle is more appropriately placed in the query display window.

5.2.2 Table Rectangles and Node Rectangle

There are two table rectangles one on the left and one on the right of the node rectangle as in figure 5.1. The display of table objects described in §6.6.1 is restricted to the table rectangles. The display of node objects described in §6.6.3 and §6.6.4 is restricted to node rectangle. While such a set up is not desirable in a diagraming tool for drawing the full schema, in a query language expression this was considered more appropriate for the following reasons.

The join nodes (§6.6.3) are used to connect tables to define access paths. With each join node there may be associated text that describes the connection. At the time of definition appropriate connecting text may be specified in both directions. By selecting the correct text a left to right reading of the connecting text can be maintained. To maintain this left to right reading the tables and nodes must be restricted to these rectangles. Abandoning this display set up will diminish the usefulness of this connecting text.

The connecting text can be presented for both directions as in an entity relationship diagram and permit a free format display without the above restriction. If a free format placement is allowed then it must be left to the user to arrange the objects for good comprehension. With a complex query if the user places tables of a query where insufficient room is left for displaying the connecting text, the comprehensibility of the query will be greatly diminished. The alternative is to automate the placement of objects and routing connections. The difficulty in implementing an algorithm for this

has been noted in [Fogg84]. Particularly in an interactive environment where response time is crucial it is important to avoid such complexity where the gains that result can be met by simple restrictions as described above.

In addition to the connecting text the node rectangle is also used for the display of subqueries (§5.10). Subqueries are displayed in reduced scale by users action to facilitate its application in the current query. A physical distinction of areas in displaying subqueries is also considered to result in more clarity.

5.2.3 Expression Rectangle

Comparisons of fields of tables are specified in this rectangle. This facility is most used in the definition of equi-joins to be permanently stored in the dictionary (§5.6). It can also be used in the query definition for comparisons of fields (§5.8).

5.3 Tool Palette

Eight tools have been added to the tool palette of the GQL system as an aid to the definition of relationships, subqueries and queries. The Icons have been chosen so as to reflect the action of the tools they represent. The Macintosh cursor that denotes the position of the mouse also changes its shape in accordance with the tool that has been selected. User actions are thus classed into the following categories for representation as a tool in the palette. In addition, a delete tool for editing and a move tool for placing objects to users preference have been included to the pallette.

5.3.1 Annotate and Deannotate

Objects of a query can be annotated and deannotated. Human capacity to assimilate information is enhanced when it is presented in layers of increasing detail. The above facility thus results in the user perception of a reduced complexity. The tools used to produce these result are

1. View tool to Annotate 
2. Box tool to Deannotate 

5.3.2 Request New Objects for Query

Requests for a new object can arise in many instances in query expression . The 'box' tool is also used for this purpose. Thus placing the 'box' tool in the appropriate location and clicking generates a new object. Since in the GQL system most objects are represented as rectangular objects the box icon is also appropriate for this action. New objects added to a query using the box icon are

1. New table.
2. New expression (This can also be generated using 'expression' tool).
3. An attribute to a table that has been previously deleted.

5.3.3 Make Connection

Actions that result in a graphical line connection can be generated using the 'connect' tool. In GQL line connections are made between two table objects denoting an association defined in the dictionary and between a table in the current query and a table in a subquery used within the current query.

5.3.4 Write Expression

All actions that produce the generation of an expression are initiated using the 'expression' tool. These are to add a qualification to an attribute of a table (§5.5) and to write an expression comparing two attributes in the query (§5.6 and §5.8).

5.3.5 Make Subquery

Selecting a section of the current query to be converted to a subquery is done using the 'query' tool. This action converts the selected section into a subquery. This is discussed in detail in §5.9.

5.3.6 Get Aggregate

Fields of a query can be selected with this tool to request an aggregate value over the filed (§5.5). Only simple aggregation is possible in the current version of GQL. Possible extensions are discussed in §5.12.

5.4 The List Dialog

A scrollable list dialogue is used through out the system for presenting the user with allowable lists of items for particular actions. This uniform approach for selecting options under various circumstances provides an interface that is less confusing to a user. The following list includes the instances where this technique is used.

Requesting a new table in the query

Requesting a new table connected to an existing table in the query

Requesting a connection between two tables

Requesting addition of an attribute to a table in the query

Requesting an addition of a subquery within the query

Requesting the subquery list for deletions

Requesting the subquery list for selecting a query

Requesting the list of possible aggregate functions.

5.5 A Single Table Query

Figure 5.2 illustrates a table representing the **item** relation. Using the 'view' tool it has been expanded to display all its fields. Examples of collapsed tables are found in figure 5.9 where tables **sale, store** and **dept** have not been expanded. If an expanded table is larger than the screen size, then the hidden parts can be brought into view by sliding the table horizontally using the 'move' tool.

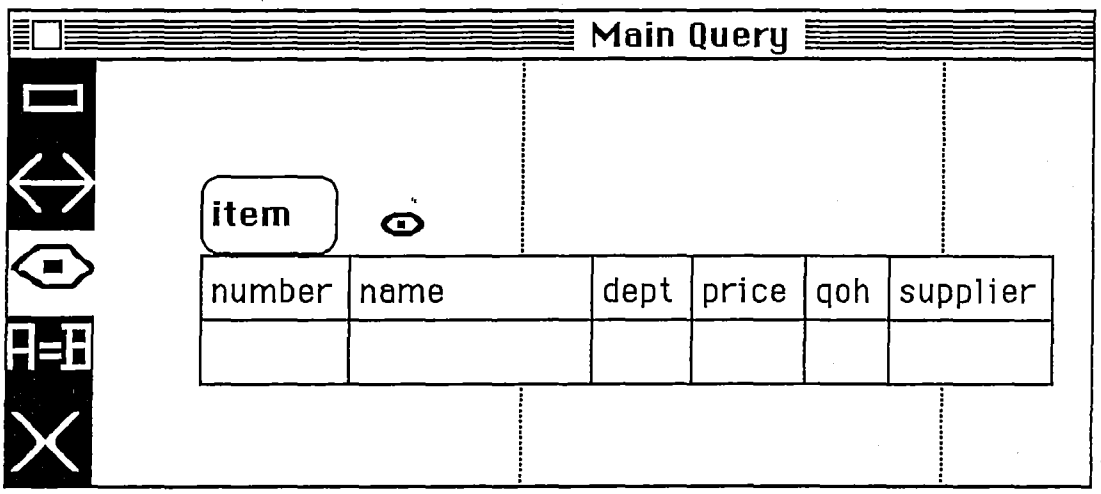


Fig 5.2

The 'delete' tool can be used to delete fields that are of no relevance to the query, thus allowing for simple display. Figure 5.3 shows the same table after some fields have been deleted. Using 'view' tool on the fields they can be selected or deselected for target list. In figure 5.3 **name** and **qoh** fields have been selected for target list.

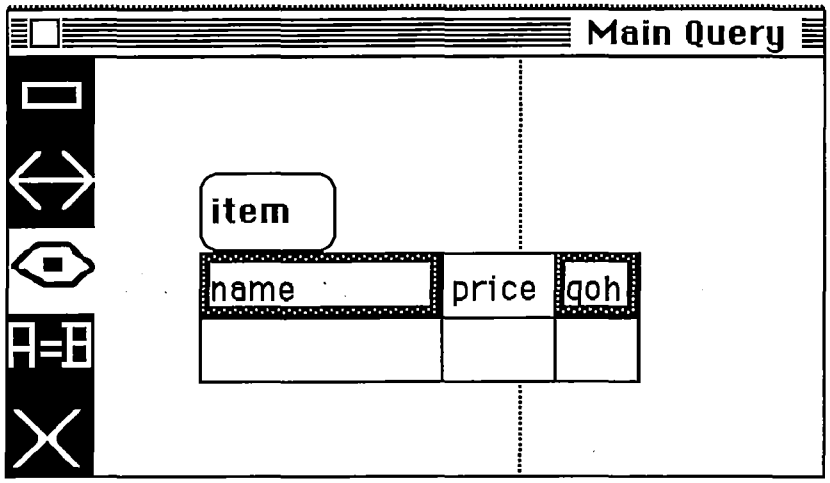


Fig 5.3

Figure 5.4 shows the aggregate function average specified as the target for the **price** field. The 'aggregate' tool is used to specify this. Any number of aggregate values can be retrieved through a single retrieval with this tool. In the figure only one aggregate has been requested.

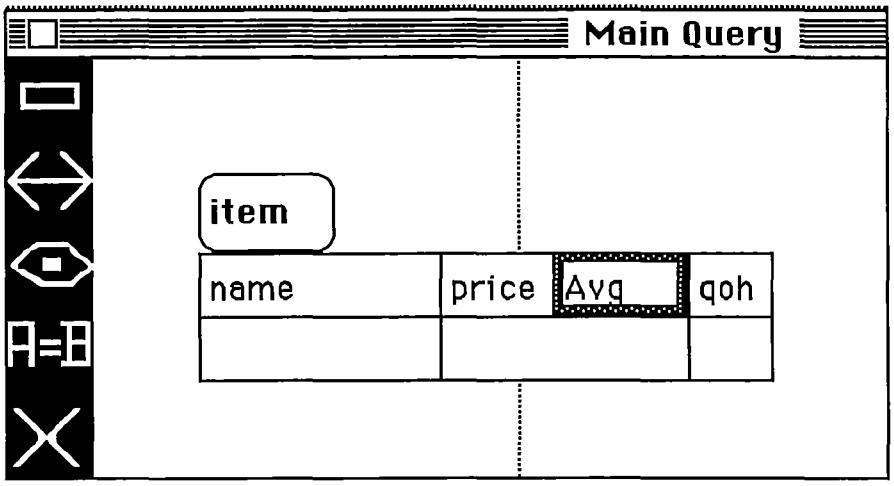


Fig 5.4

Qualifications are placed on fields using the 'expression' tool on the row displayed below the field headings. The display in figure 5.5 is presented for specifying the expression. When this display is dismissed by the user the expression appears in the table display as in figure 5.6. The qualification 'price > 1000' and 'qoh < 5' has been placed on the fields of the table **item**. Thus the tabular method is used as specified in §1.3.3.2. The full tabular technique has not been implemented in the current version. Only single conditions can be placed on the fields as in figure 5.6. As described in §1.3.3.2 multiple rows can be used to specify ORing of selection conditions.

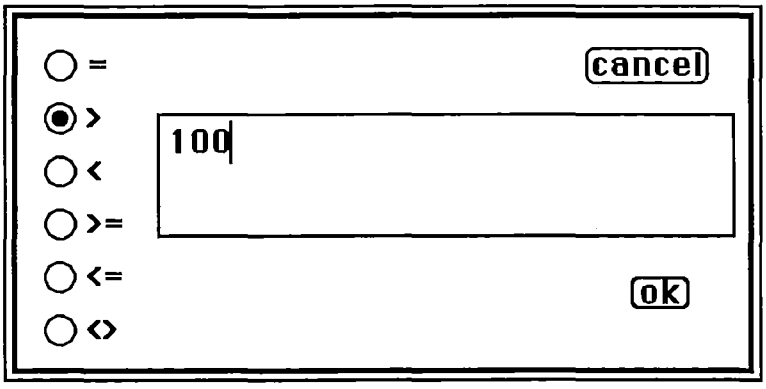


Fig 5.5

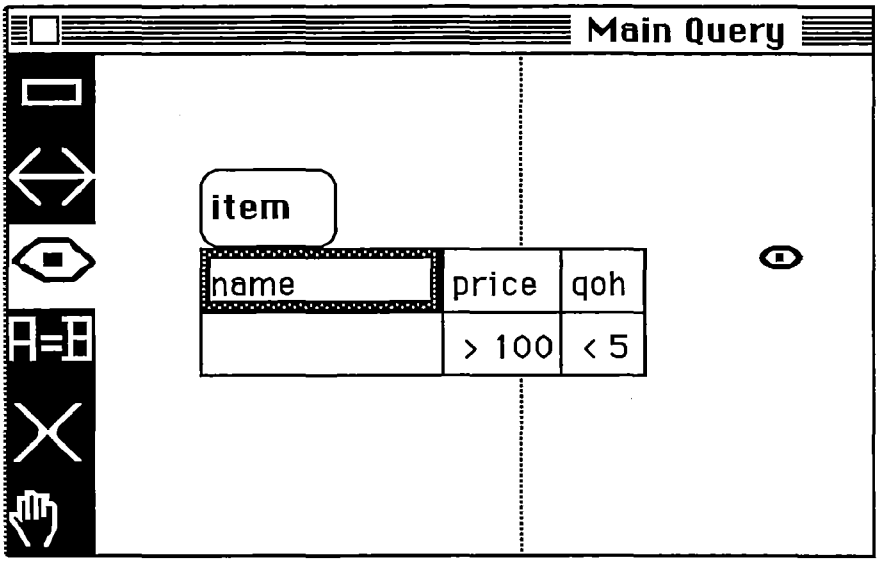


Fig 5.6

5.6 Defining Relationships

Figure 5.7 shows the query window when the menu option 'Define Rel-ship' is issued. In this mode relationship involving equi-joins are specified. The equi-joins are built in the expression rectangle (§5.2.3) using the expression tool. More about building expressions is found in §5.8.

When user closes the window after going into 'define relationship' mode the display in figure 5.8 is presented for specifying connecting text for the relationship. The above definition is now permanently stored and can be used for query building.

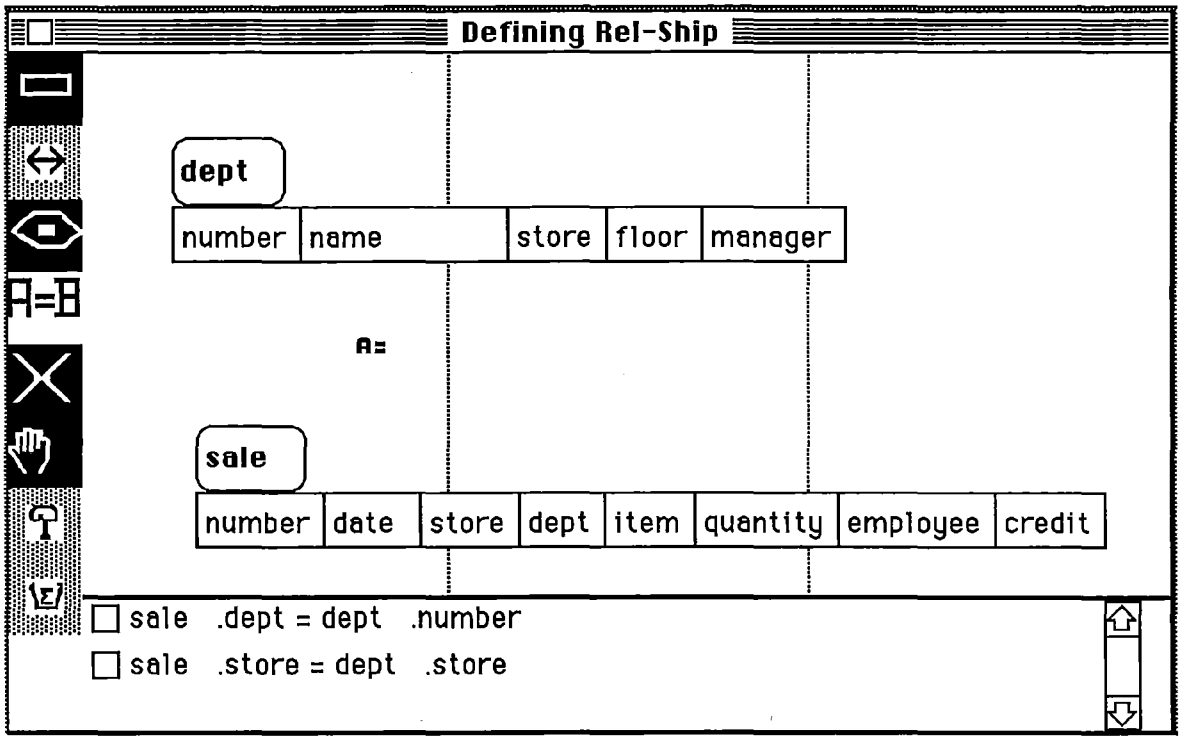


Fig 5.7

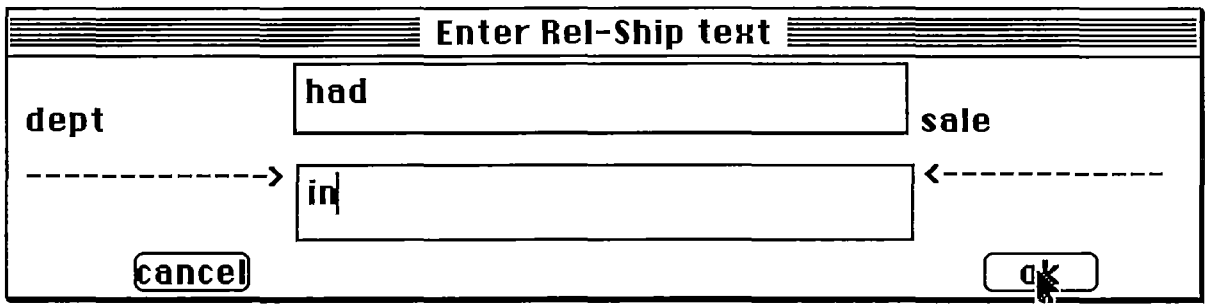


Fig 5.8

5.7 Query Using Relationships

Figure 5.9 shows a query that uses relationships that have been defined on the entities. Definition of one of these relationship is shown in figure 5.7 and 5.8. Using the 'connect' tool on a table in the query will present all tables that have a relationship permanently recorded to this table. The list dialog of §5.4 is used to present the options. The conditions that must be applied to the tables to effect the relationship are thus replaced by the connecting text in the node rectangle.

Any of the tables in a query can be expanded with the 'view' tool. In figure 5.9 the **item** and **employee** table have been expanded.

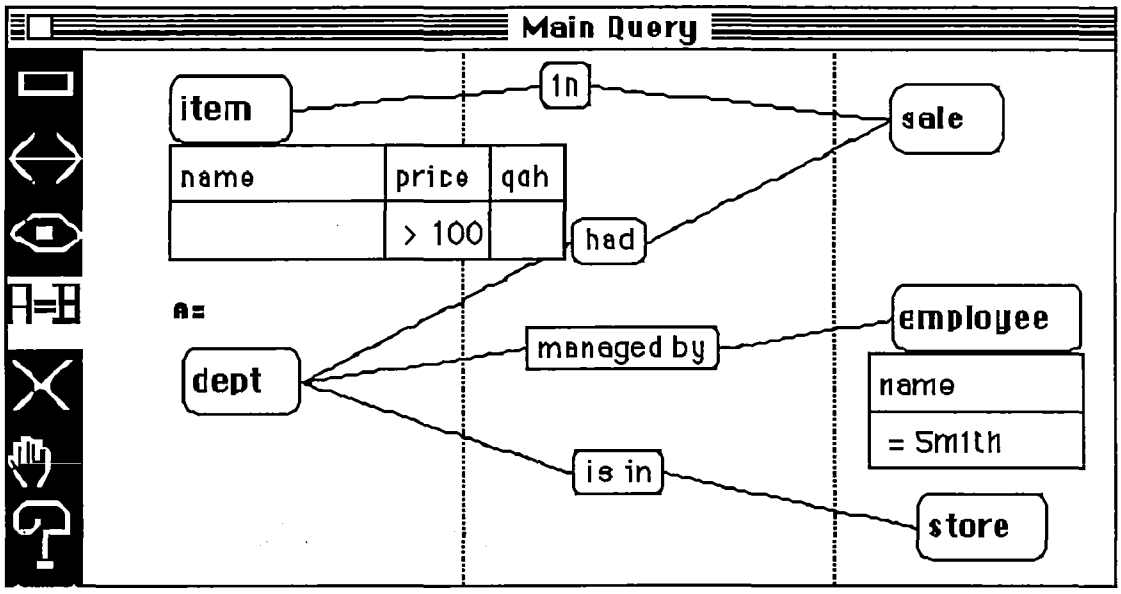


Fig 5.9

5.8 Two Table Query Using Expression Rectangle

Figure 5.10 shows a query that uses the expression rectangle to specify comparisons of attributes in a query. A paper and pencil mode of writing expressions is adopted in designing the actions of the user needed to construct these expressions. Unlike the function tile technique described in §1.3.3.1, in a paper and pencil mode the bracketing convention must also be included for writing complex expressions. In the GQL system only simple comparison expressions can be constructed.

Using the 'expression' tool as pen the expressions can be constructed. Thus a qualification that compares two fields from tables can be constructed by actions similar to writing with a pen. Editing is provided through a reset button for each expression. Clicking the 'delete' tool in the the reset button of an expression will erase the expression.

Unlike in the above section, where only equality of fields can be specified for defining relationships, in query mode the expression rectangle can be used to specify any scalar comparison of fields.

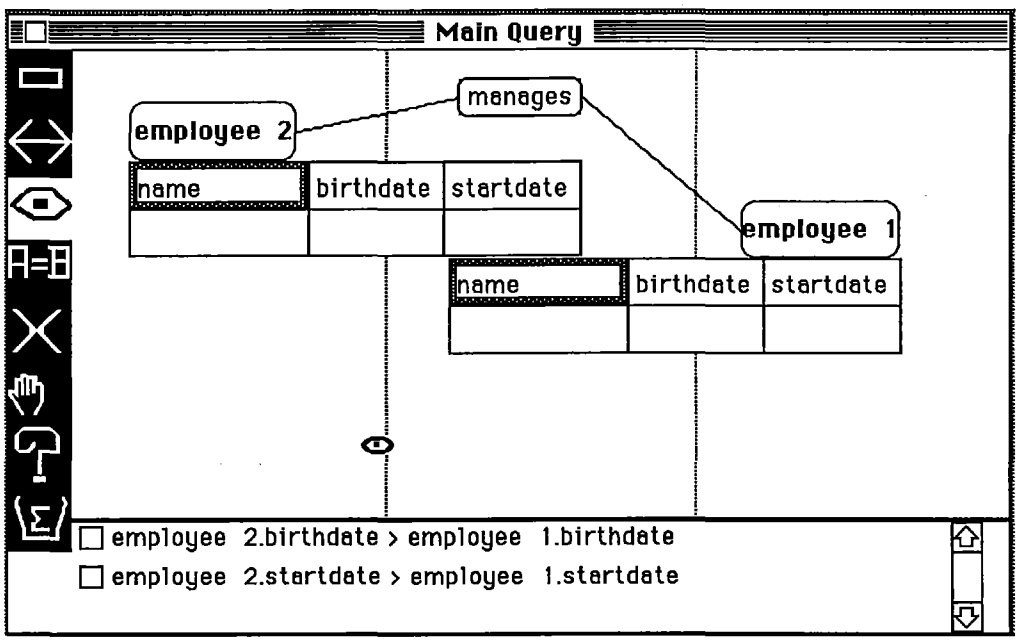


Fig 5.10

5.9 Named Query Definition

A subquery may be defined by either issuing the 'New Subquery' menu option or by selecting a section of the current query with the 'query' tool. When the 'query' tool is used on an object of the query the section of the query that is connected to this object is highlighted as in figure 5.11. The table **store** in figure 5.15 is not highlighted since it is not connected (§3.2) to the object selected which is the table **employee**. The display in figure 5.12 is presented to enter a unique name for the subquery to be formed. When a subquery is defined with this method, in addition to defining the query, an instance of the subquery is added to the current query. Figure 5.13 shows the half scale display of the subquery that was defined, after some parameter instantiation has been applied. The technique for applying parameter instantiation is

described in the §5.10.1. Figure 5.14 represents a completed query that uses the subquery defined above. Equivalent QUEL translation of this query can be found in §7.2.3

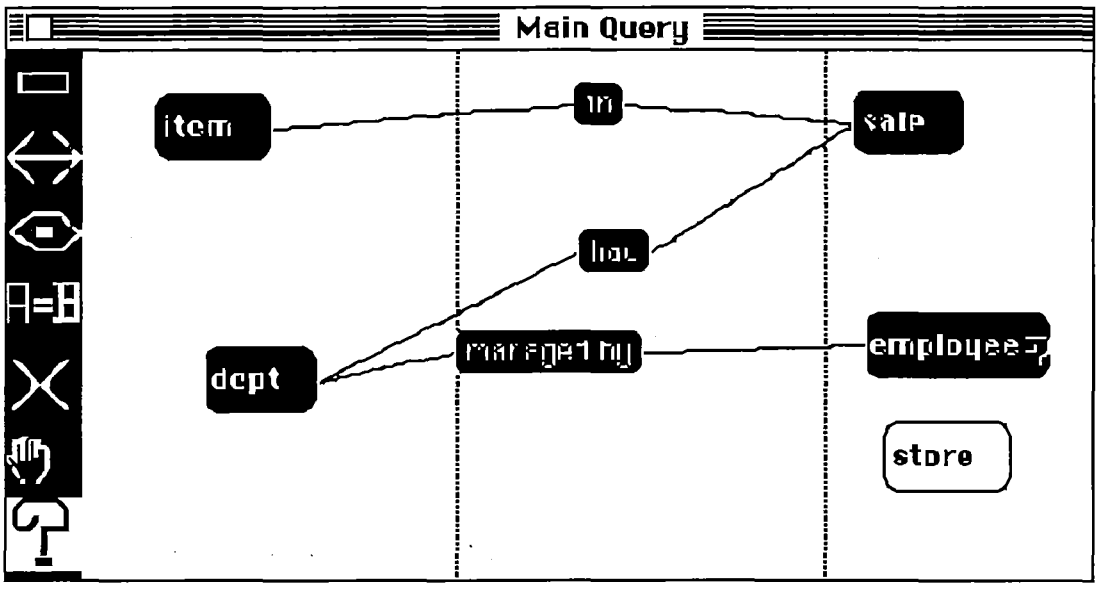


Fig 5.11

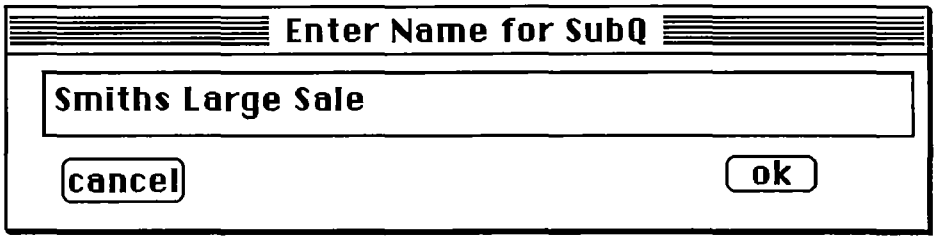


Fig 5.12

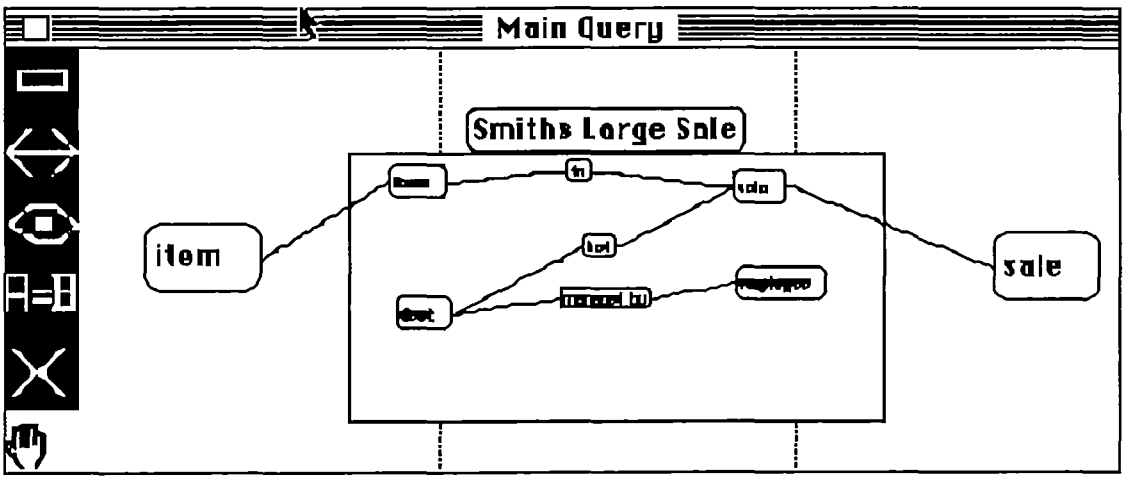


Fig 5.13

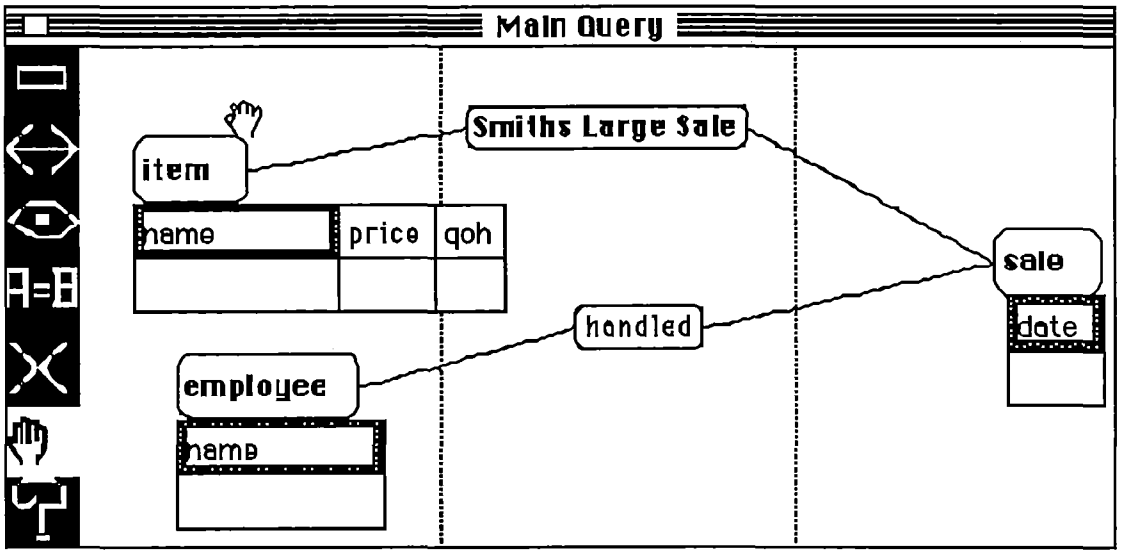


Fig 5.14

5.10 Subquery Display Within Another Query

By permitting definition and use of parameterised queries the advantages that result from the use of procedures in 3GL languages can be achieved in a query interface. In standard query languages, such as QUEL, this is permitted through the use of macros. In GQL a graphical technique is adopted for using parameterised query. Thus all the advantages of this ability is provided with a much simpler interface.

In a text based interface modules are defined and used in other modules through the use of parameters. A module can be used within another module more than once as shown below

Main Module
.....
M1(A, B)
.....
M1(C, D)
.....

where M1(P1, P2) is a the module with two parameters P1 and P2. Thus in a text based interface using the module more than once necessitates its appearance more than

once. In a graphical interface it is feasible, and probably desirable, to display the module only once but use it in more than one instance. Thus there are three possible ways of applying a module graphically.

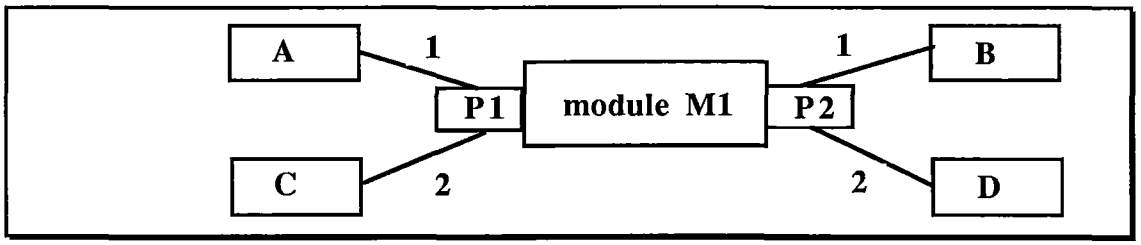


Fig 5.15

1. Single display of module and multiple parameters. As illustrated in figure 5.15 if the number of parameters of the module is more than one some way of graphically grouping the parameters of one instance of application is necessary.
2. If the module is restricted to have only one parameter this additional grouping problem does not exist.
3. Follow the text based style and display the module as many times as it is applied.

The first alternative requires too much detail to be displayed graphically. The second and third alternatives are both suitable for the GQL system subquery application. Taking the second alternative means a subquery can only be applied through one table. In most instances this semantic interpretation of subquery application would be valid. The implementation of GQL takes the third approach. In general a subquery may have any number of tables all of which are potential parameters.

A subquery can be added to a query either by issuing 'Add Subquery' menu option or by converting part of the current query into subquery as described in §5.9. Using the

'view' tool a subquery can be be displayed in half scale for applying parameter instantiations. By using the 'view' tool on the subquery displayed in half scale, the subquery can be viewed as the current query in the query window. More on subquery traversal is found in §5.11. The 'box' tool can be used on a subquery object to collapse it as shown in figure 5.14, where the subquery **Smiths Large Sale** has been collapsed.

5.10.1 Parameter Instantiation

Parameter instantiations are applied using the 'connect' tool. With the connect tool a table in the subquery is selected and dragged to a table in the main query. A line starting from the selected table is drawn that traces the mouses movement until it is released. If it is released in a compatible table in the main query then a parameter instantiation is established between these two tables. This is displayed by drawing connecting lines.

The instantiations can be deleted with the 'delete' tool on the appropriate table in the half scale subquery display.

5.11 Traversing the Subquery List

A single query display window is used to view/edit any of the subqueries in the dictionary. For reasons of orientation of the user one query, considered as the main query, is automatically defined in the GQL system. Thus whenever the user closes the query display window currently displaying a subquery the system returns to the main query. To emphasize this changing view a dynamic visual clue is presented before the change. This takes the form of a continuously decreasing rectangle display that takes about 3 seconds to complete.

In addition to query display this same window is also used to add a relationships to the dictionary. Since the interface required for this is a subset of the query definition

interface this is again considered an appropriate technique. During this phase irrelevant parts of the query definition tools are inactivated.

Thus the changing view in the query display window can be effected in the following ways.

1. Annotation of a subquery node in the current query displays the query associated with the subquery node.
2. Selecting 'New Subquery' menu. This will display an empty query for constructing a new query.
3. Selecting a subquery from the subquery list presented when the 'Select Subquery' menu option is chosen.
4. Go-Away box of the query display window closes the current query being viewed and returns to the main query(home base). By maintaining a stack of subqueries that have been traversed a backtrack facility can be provided for returning to the earlier subquery rather than returning to the main query.

5.12 Result Display

Result display takes the form of a separate window in the Apple Macintosh interface. The results are presented in a tabular form which can be viewed by vertical and horizontal scrolling. Thus results retrieved from many queries can be viewed through this single window. The size of the buffer used limits the amount of information that can be viewed. Buffer size is set to 5K bytes in the current version. The buffer is used in a cyclic mode, thus retaining the last 5K characters. The size of the buffer chosen is sufficient to hold the result of most ad hoc queries issued by the user of such an interface. However it is feasible to allow the user to set the result buffer size through a configuration menu in a future version.

Figure 5.16 shows the results of the query in figure 5.14. The domain characteristics in the local dictionary can be used if available to present the table in a standard format

such as titles for columns. Currently the width of the domain is used to determine the table set up.

The result buffer is also used to view the QUEL equivalent of a GQL query by issuing a 'Translate' menu option. The QUEL equivalent of the query is output to the result buffer. With this facility, GQL can be used as an educational tool. GQL can be used to teach the more difficult languages such as QUEL by allowing the students to build a query in GQL and view its equivalent translation to QUEL.

The results retained in the result buffer can be saved to a file as ascii characters by issuing the 'Save Result' menu option, thus the size of the result buffer also limits the size of the saved file. The query translation given in §7.2.3 was obtained by issuing a 'Translate' menu option which outputs the QUEL translation to the result buffer and then issuing a 'Save Result' menu option to put the result out to a text file.

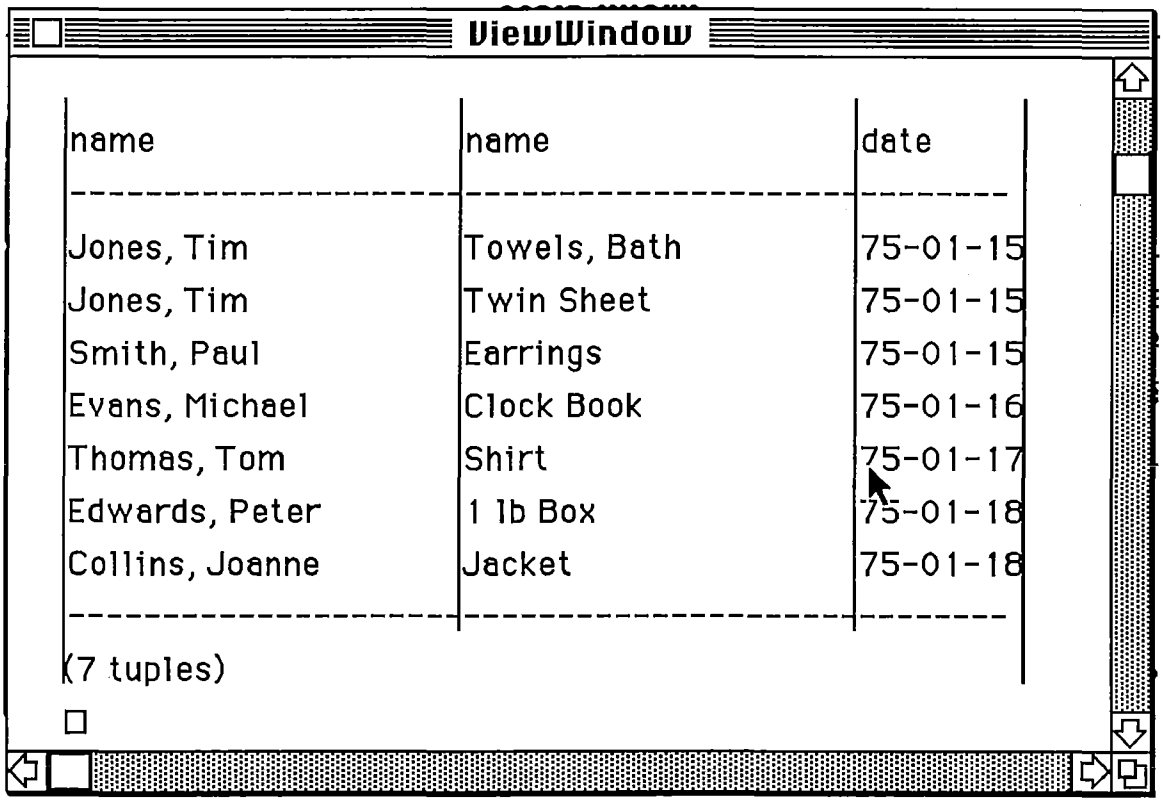


Fig 5.16

5.13 Extension of GQL's Graphical Techniques

The GQL representation can be used to express aggregate conditions and aggregate target fields in a manner similar to the tabular technique used for expressing conditions and target fields for tables of the query. The relationship connecting two tables can be converted into an aggregate table (§4.5.2). Which of the two tables in the association is to be grouped can either be deduced by storing the cardinality of the association in the dictionary while creating the association, or specified by the user building the query. To translate the above query into a valid relational query in SQL a hierarchical structure for tables in the query must be enforced (§3.4).

Chapter 6

Dictionary Support in GQL

6.1 Introduction

This chapter gives a detailed description of the GQL local dictionary. In a friendly interface to relational databases the need for additional layers of information on top of the relational schema description was discussed in chapter 4. These information layers can be maintained either as additional schema relations in the host system or as a local dictionary in the workstation or distributed between the two. Certain information such as relationships that describe the global schema are best maintained in the host . These should be entered into the host schema using tools such as data modelling tool. The current implementation of GQL is a retrieval based interface and does not provide any mechanism for changing the host schema. The GQL system maintains its own local dictionary which is initially built with information from the host database system. It provides the facility for adding further information layers that are useful in query building. All the information layers that are built using the GQL interface are maintained in the local dictionary.

GQL can open one dictionary at a time during execution. The dictionary must be read into memory at the start of a session. Dictionaries may be opened and closed during session to access different databases.

The data structures used for the implementation of the dictionary are described in Appendix B. An estimation of the memory requirement for maintaining it in memory during run time is also given in this appendix.

6.2 Adding Information Layers to Host System

The option of adding information to the host schema instead of maintaining a local dictionary as in GQL is discussed in this section. The following disadvantages can be noted, if GQL is to add the information to the host schema. The approach requires slightly greater traffic between the host system and the workstation as the required local structure for GQL is built every time by extracting information from the host. The host schema administration must take these extra information layers as part of its responsibility thus requiring greater host system changes before GQL front end is usable. The front end itself will require greater host system dependent development work. The subquery features that are described in §6.6 need not be considered as semantic information but as an aid to user; thus it is inappropriate to bring them under the host system administration. The workstation is a more appropriate location to maintain subqueries.

In an ideal situation the access path information etc should be under host system administration while users of the front end should have the freedom to maintain additional access path information and subqueries locally for their convenience. This is more so when the database has a very wide class of users. Thus under the current relational systems it is considered a local dictionary structure is more appropriate.

6.3 Three Layers of Information in the Local Dictionary

The local dictionary maintains three layers of information where the layer at next level uses the structures in the previous levels for its own definition. All queries can be formulated using the first layer alone. Further layers are added as aids in query formulation. These layers are described below. The structure of the dictionary is first presented as a set of relations. The attributes in italics form the *key fields* of the relation. Underlined fields are *foreign keys* in the relation. A diagrammatic representation of the dictionary is presented in figures 6.1 and 6.2.

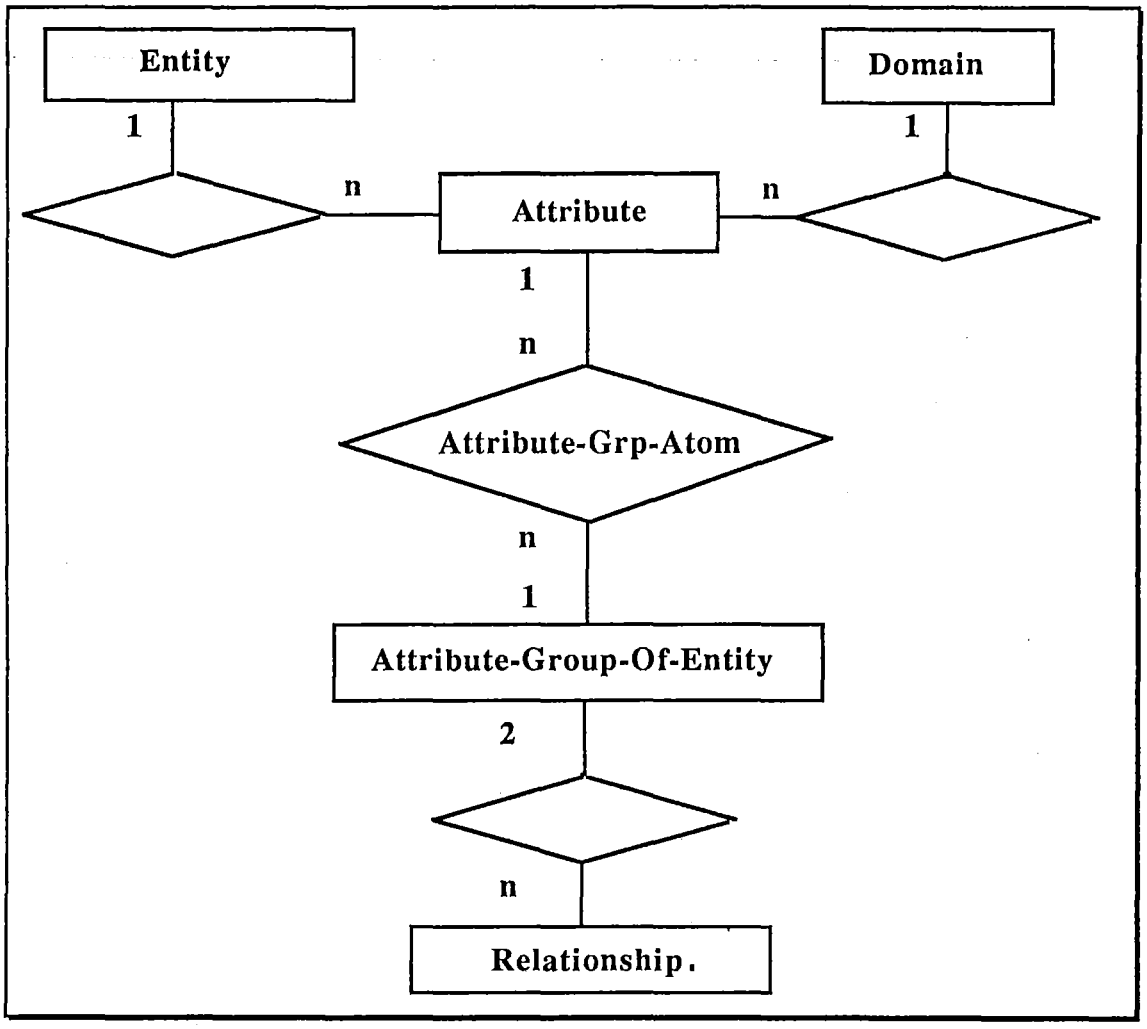


Fig 6.1 First and Second Layers of Dictionary Structure

6.4 Initialising the Local Dictionary-First Layer

Any relational schema will maintain the list of relations and the fields in the relations. This is the minimum information that must be drawn from the host system by GQL system.

6.4.1 Domain Definition in Relational Database

The definition of a fully relational system (§2.2) requires the domain structure to be more precisely defined. One important reason for this is to maintain the referential integrity rule. A true domain structure represents important semantic content of a database schema. This is evident from the discussion in §4.2 which suggests that most

access path information can in fact be deduced from a precisely defined domain structure. Most real world attributes can be specified to have special properties other than the general types of integer, real and character. Such specialized properties can also be useful in maintaining integrity during updates of these fields. Recording such information in the host system schema can also be a way of communicating this to all users. A domain structure is more appropriate to record this rather than having to repeat it with every attribute of the relations. This information is very valuable during query writing when selection conditions are to be placed on fields. This is especially so when the domain is restricted to having a finite set of values such as 'yes', 'no' fields. In such cases possible values can be presented to the user for selection. The local dictionary of a workstation front end when used for report writing tasks can make use of these additional properties of attributes such as presenting fields in standard styles.

Current systems including INGRES do not allow user defined domains. In INGRES every field in a relation is given a format descriptor which identifies it as integer, real or character and contains its field width. In the GQL system interfaced to university INGRES this format descriptor field of the attribute is treated as a domain identifier and used to create the domain objects of the local dictionary. This however does not truly describe the domain structure of the database schema. One instance where the GQL system uses this information is to enclose attribute values of character types in quotes as required by QUEL. The domain characteristics in the current implementation only describes the integer, real and character types. But the Domain Characteristics attribute above may be expanded for more specific type description. The following relations represent the local dictionary structure described so far.

Relation	**Attributes**
Domain	*Domain-Name* Domain-Characteristics
Entity	*Entity-Name*
Attribute	*Attribute-Name* *Entity-Name* Domain-Name

6.5 Relationships - Second Layer

This layer consists of the definition of equi-joins between two base entities. When a foreign key in an entity is a composite attribute, the equi-join will involve equating more than one set of attributes from the two entities. A structure for grouping attributes into composite attributes within an entity is therefore required to specify a generalized equi-join. The term 'attribute group' is used in this work to refer to a group of attributes as defined in figure 6.1. The reasons for this are purely historical since this term is used in the implementation code of GQL. Thus by adding as many equi-joins as required in this layer an effective access path definition aid is provided to the user. The current GQL dictionary structure does not distinguish between the different types of base entities such as those in the entity relationship model or RM/T model. The GQL features provided in the current implementation are not enhanced by the addition of these details. The cardinality involved in the equi-joins is also not recorded in the dictionary. This is discussed in §4.5.

To each equi-join a linking text is added. This serves the following purposes:

1. When the user is presented with the list of equi-joins that involves a given entity this text is added to the list appropriately, thus giving additional explanation as to the meaning of the relationship.
2. In query display this text is used to graphically link the two tables involved, thus improving the comprehension of the query.
3. In applying a subquery as illustrated in §5.10 this display is used in linking a subquery table to the current query table, the connecting text plays an important role in this process.

The set of relations that represents the relationship between entities in the local dictionary is listed below.

Relation	**Attribute**
Attribute-Group-Of-Entity	*Attribute-Group-Name* *Entity-Name*
Attribute-Grp-Atoms	*Attribute-Group-Name* *Entity-Name* *Attribute-Name*
RelationShip	*Attribute-Group-Name1* *Attribute-Group-Name2* Relation-Text1 Relation-Text2

6.6 Subqueries - Third Layer

The use of predefined subqueries as an aid to user interface is clear. The definition of equi-joins in layer two can be considered as a subquery definition where only equi-join conditions are allowed. At the early stages of design the possibility of treating equi-joins too as a subquery, and thus allowing for only one layer for equi-joins and subqueries, was considered. However this approach deviates from the trend in semantic modelling, and was thus rejected. The advantages of the GQL subquery facility are:

1. Named subqueries stored permanently in the local dictionary are tools for greater efficiency for a frequent user, eliminating the need to formulate queries every time. Non expert users can still use queries predefined by someone else without having to request others to do the retrieval for them. Enabling such users to make simple modifications to queries thus providing a degree of flexibility.
2. Use of named subqueries in formulating other queries can be used to maintain repeatedly used query sections which can then be used in the formulation of other queries without having to be concerned about its details each time. The subquery structure can be used to hide away the details of sections into modules thus enable clear query expression. This is illustrated in §5.10.

Any query can be stored away in the dictionary as a named query. The structure of a query and its representation in the local dictionary are described below. In the relations used to describe this layer, surrogate values are used to reference objects in the first and second layer of the dictionary. In the actual implementation of the dictionary, surrogate values are assigned to many objects for performance reasons and these are described in detail in Appendix B. Following sections describe the structure used to represent a query in GQL.

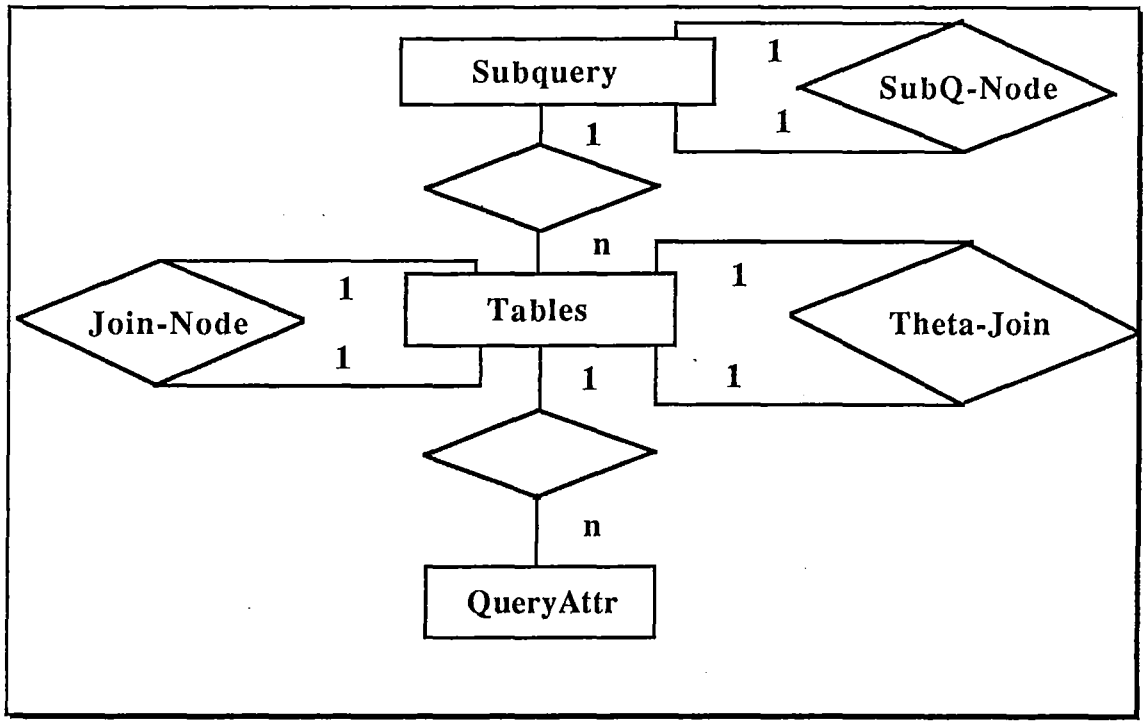


Fig 6.2 Third Layer of Dictionary Structure

6.6.1 Tables of a Query

A table of a query ranges over a base entity recorded in the local dictionary. The base entity in the GQL local dictionary may well be a view defined in the host system. The tables are given the names of the base entity as recorded in the local dictionary as its identifier. If more than one entity from the same base entity is used in a query then these tables are numbered so as to give them a unique identifier. Thus a table in a query

represents a set of tuples satisfying the conditions that are placed on it. A table in a GQL query thus is same as the table concept defined in chapter 3. The aim is to use Tabular techniques as discussed in §1.3.3.2 over these tables to express conditions. The tables of the dictionary are represented by the following relation.

Relation	**Attributes**
Table	*Tuple-ID*
	Entity-Surrogate

6.6.2 Attributes of a Table

The attributes of a table can be represented as a one to many relationship. A table when first generated by the user carries all the attributes of the entity from which it derived. The user may delete the attributes that are of no interest from the table. The purpose of this is only to remove unnecessary information thus having a less cluttered display. This operation is not equivalent to the projection in algebraic languages. Conceptually the table derived from an entity of the schema will always have all the attributes of that entity. Deleted attributes can be added back to the table for redisplay if the user wishes. The attributes of a table in the dictionary are represented by the following relation.

Relation	**Attributes**
QueryAttr	*Tuple-ID*
	Entity-Surrogate
	Attribute-Surrogate
	Attribute Qualification
	Compare Operator

6.6.2.1 Qualification of Attributes.

Selection conditions can be placed on the attributes within a table following the tabular convention. Current implementation permits only simple scalar comparisons to be placed on attributes. Thus in the current version the above relation includes *Attribute Qualification* which is a character string and *Compare Operator* together representing the qualification placed the attribute. This structure needs to be expanded further to represent the full tabular representation of qualification.

6.6.3 Selecting Access Path - Join Node

Previously defined binary relationships recorded in the local dictionary can be used to connect two tables in a query. This connection is represented as a join-node in the query. Typically the connected query concept of §3.2 is established using nodes of this type in a query. This is represented by the following relation in the dictionary.

Relation	**Attribute**
Join-Node	*Tuple-ID1*
	Entity-Surrogate1
	Tuple-ID2
	Entity-Surrogate2
	RelationShip Surrogate

6.6.4 Subquery Node

This represents a predefined query within another query. When expanded the subquery node will reveal a half size graphical version of the query that it represents. A table in the subquery node can be connected to a table of the same entity in the current query effectively equating the two. Only a single connection is possible from one table in a subquery node. However many such connections can be made from separate tables in the subquery node. A subquery can be defined using any number of subquery nodes. A predefined subquery can be used in the definition of any number of other queries. This is illustrated in §5.10. Some integrity problems that must be avoided within this feature are discussed in §7.3. The following relations are used to represent the use of a predefined subquery within another query.

Relation	Attributes
SubQ-Node	*SubQ-Name* *Occurrence-Number*
SubQ-Node-Connect	*SubQ-Name* *Occurrence-Number* *Entity-Surrogate* *Tuple-ID* *SubQ-Tuple-ID*

6.6.5 Theta Joins

Comparisons of two attributes of the tables in the query can be made using this feature. The use of this is illustrated in §5.8. In the GQL implementation the definition of relationships to be stored in the dictionary as such is specified using this feature as illustrated in §6.6. During the relationship definition mode this construct is restricted to specifying equality condition only. During query formulation this feature allows the specification of any type of scalar comparison. The following relation is a representation of this in the dictionary.

Relation	Attribute
Theta-Join	*Tuple-ID1* *Entity-Surrogate1* *Attribute-Surrogate1* *Tuple-ID2* *Entity-Surrogate2* *Attribute-Surrogate2* Compare-Operator

Chapter 7

Design and Implementation Issues

7.1 Introduction

In this chapter some of the design and implementation issues that were resolved for GQL implementation are described. Notably the algorithm for translating a query held internally in GQL to a QUEL query is described. Some of the integrity problems that arose as a result of maintaining subquery list and how these have been resolved are given. The algorithm to establish a **connected** query as defined in §3.2 is also presented.

7.2 Query Translation

A query is held in the GQL system using the structures described in the preceding chapters. The following algorithms are used to translate a GQL query into an equivalent QUEL query. In QUEL translation a table that is connected to a subquery table is treated as the same tuple variable. The main algorithm used for QUEL translation is listed §7.2.1.

For each subquery tuple variables have to be regenerated twice in the main algorithm given in §7.2.1. Once to generate the range statements and again to generate qualification. This is necessary since the same subquery may be used more than once in a query definition. Therefore tables in a subquery may appear more than once as tuple variables in the translated query. The algorithm to generate tuple variables is given in §7.2.2. The routine is called with the subquery usage details which will be null for the main query that is being translated and provide details of parameter instantiation if it is a subquery used within another query.

7.2.1 Main Procedure algorithm

```
If Query Is Connected Then

  If Target Fields Specified Then

     begin

             Generate Range statement      {this is a recursive
                                           procedure that generates
                                           range statements for all
                                           the subqueries contained in
                                           it }

             Generate Target Lists         {target list fields can be
                                           only from the main query}

             Generate Qualification        {this is a recursive
                                           procedure that generates
                                           qualifications for
                                           subqueries contained within
                                           it}

             Receive Host

End
```

7.2.2 Algorithm to Generate Tuple Variable Name

```
For each Table in Query

     If Usage details is not null

        Then                               {the generation is for a
                                           subquery appearing in
                                           another query}

        If connected to a table in the query where it is used

           Then use that table tuple variable

           Else Generate tuple variable

     Else Generate tuple variable          {the query is main query
                                           and tuple names must be
                                           generated}

If the step was to Generate tuple declaration

     Then Generate tuple declarations

End
```

7.2.3 A Translated Query

```
Range of T01S01 is employee                 {employee in the main query
                                            whose name is selected as
                                            target}

Range of T02S01 is item                     {item and sale in main
                                            query is connected to item
                                            in subquery, therefore only
                                            one tuple variable to
                                            represent both instances}

Range of T03S01 is sale

Range of T01S02 is employee                 {employee and dept in the
                                            subquery are not connected
                                            to tables in main query,
                                            therefore tuple variables
                                            need to declared}

Range of T02S02 is dept                     {dept in the subquery}

Retrieve (

T01S01.name

 , T02S01.name

 , T03S01.date

 )

Where

  T01S01.number=T03S01.employee             {relationship employee
                                            handles sale}

 AND (T01S02.name = "Smith")

 AND (T02S01.price > 100)

 AND T01S02.number=T02S02.manager           {relationship employee
                                            manages dept}

 AND T02S02.number=T02S01.dept              {relationship dept had
                                            sale}

 AND T02S02.store=T03S01.store              {second conditions for the
                                            above relationship}

 AND T02S01.number=T03S01.item              {relationship item in sale}
```

The query graphically represented in figure 5.14 is translated into the above QUEL query by the translation algorithm. The comments next to the QUEL statements are not output by the GQL system. The numbering T01 onwards are used to name tuple

variables within a query, where as the numbering S02 onwards uniquely identifies the occurrence of a subquery within another query. Thus S01 represent tuples in the main query being translated, while S02 represent the subquery 'Smiths Large Sale'.

7.3 Integrity Maintenance in the Subquery List

Each subquery in the GQL system is given a surrogate value by which the system references it. Users can give names to subquery by which they can identify it. A subquery can be used in the definition of another subquery. The concept behind it is described in §6.6.4. The graphical display of this usage is described in §5.9. Two integrity problems associated with subquery module definition and usage have to be resolved in the GQL system.

7.3.1 Recursive Subquery Definition - Integrity Problem 1

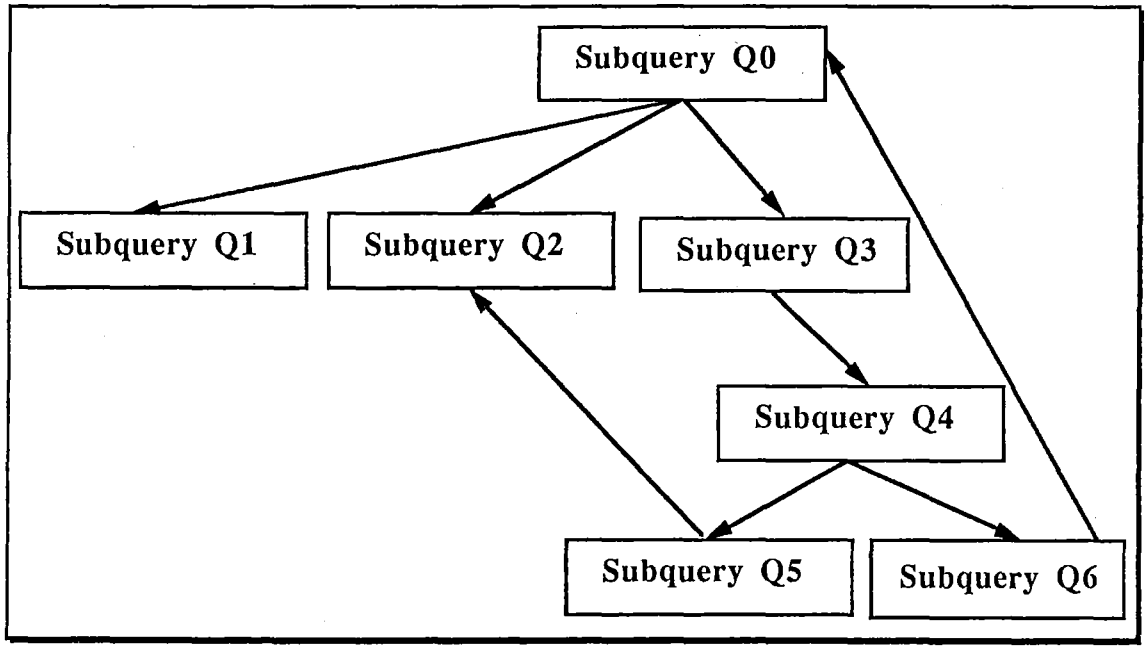


Fig 7.1

In figure 7.1 the subquery usage in other query definitions are represented by the arrows. For example Q0 is defined using queries Q1, Q2 and Q3. If unrestricted editing of these queries is permitted then Q6 can be edited to include Q1. As seen in

figure 7.1 this results in Q0 being defined in terms of Q3, Q4 and Q6. But definition of Q6 uses Q0. Therefore Q0 is recursively defined which is meaningless. However editing of Q5 to include Q2 does not create this problem.

One solution to this problem is to permit such editing and leave the responsibility to the user to maintain integrity. This would be similar to programming language approach adopted in compilers, where recursive definition is permitted. But unconditional recursive calls will leave the program in a permanent loop. It is the users responsibility to write code so that this does not occur. This approach is not suitable for an interface like GQL. On the other hand detecting the recursive definition of subqueries means an exhaustive search of the subquery list whenever a subquery is edited to add a subquery node.

The GQL implementation handles the problem by maintaining a 'usecount' in each subquery. This is an integer value that stores the number of subqueries that are defined using this subquery. Adding new subquery nodes is not permitted in subqueries whose 'usecount' is non zero. In effect this ensures an order of definition of the subqueries where used subqueries are to be defined first before their usage. This restriction in editing subqueries is only for adding new subquery nodes. Other forms of editing can still be done even when 'usecount' is non zero.

7.3.2 Subquery Parameter Invalidation - Integrity Problem 2

The considerations of this integrity problem also lead to another restriction on editing subqueries. The subqueries are parameterised by the tables in the subquery. Deleting a table that is used as a parameter in the definition of another query will result in invalid definition. A parallel problem does not exist in text based interfaces since the number of parameters in such interfaces must be predeclared as in the forward declaration of procedures in programming languages. One possible solution is to restrict the editing of subqueries with 'usecount' further. That is to disallow deleting of tables from

subqueries with non zero 'usecount'. A more relaxed restriction can be allowed if a 'linkcount' is also maintained in the tables of subqueries. This will store the number of instances where the particular table is used as a parameter in the definition of another subquery. A table is deletable only is 'linkcount' is zero. In GQL system the first approach is taken . Thus a subquery with non zero 'usecount' cannot be edited to:

1. To add a new subquery to it
2. Delete a table from it

7.4 Checking for Connected Query

A query is accepted for retrieval only if it is connected as defined in chapter 3. The same connectivity principle is used to highlight a connected query when the user selects a query object with the 'query' tool as in figure 6.11. The highlighted connected query can be converted into a subquery by the users actions. Thus it is necessary to establish a connected query for the above two purposes.

In the GQL implementation the tables of a query can be connected by any of the following query objects.

1. Join nodes representing relationship
2. Theta-Joins
3. Subquery node

If subquery nodes are to be treated as connecting nodes then the subquery itself must be a connected query. The following algorithm checks if a query is connected by assuming that the subquery node represent a query that is connected. To fully assert that the query is connected, the subqueries used in the query are checked in a similar manner for connectivity. Thus checking connectivity is a recursive procedure that checks the connectivity of the subqueries used in the query as well. The following algorithm is used in the procedure to separate a query into separate queries each of which is a connected query.

```
For each connecting node in the Query      {these are Relationship
                                           Node, Expression and
                                           Subquery Node}

    Get a Table involved in the connecting node

    If the owning query of this Table

       is the query being separated

       Then                                {this table has not been
                                           processed yet. At this
                                           stage a Table may have been
                                           processed if the Table is
                                           also involved in another
                                           connecting node that has
                                           been previously processed }

          Transfer the Table into a new subquery

          and call it FSQ

       Else set the subquery containing the Table to be FSQ

                                           {this will happen if the
                                           Table has been processed
                                           and as a result transferred
                                           into a subquery}

    For all other Tables involved in this connecting node

    If the Table is in the query being separated

       Then transfer it to FSQ

       Else merge the owning query of this Table with FSQ

                                           {this happens when this
                                           table has been transferred
                                           to a separate query through
                                           its participation in
                                           another connecting node
                                           that was processed in a
                                           preceding pass through this
                                           loop}

    For all Tables still remaining in the query

                                           {these are not connected to
                                           any Nodes}

Transfer them into separate subqueries.
```

If number of queries that the query has been separated into is greater than one, then the original query is not connected as defined in chapter 3.

Chapter 8

The Host System and GQL

8.1 The Host System Process

The design of the GQL system has taken advantage of well established theory of relational databases and their query languages. Most design details of the GQL system are independent of a particular database system. Implementations of relational systems do vary in certain aspects and some modules in the GQL system are therefore dependent on the particular host system. These modules are

1. Initialising the local dictionary
2. Translating the query into a host system query language.

The process at the host end that receives requests from the graphical front end may be an existing module in the host system or a process specially written to handle requests from the graphical front end. A special purpose process can be written so that the user at the graphical work station has a greater control over this process. For example in INGRES an EQUEL retrieve statement embedded in a host programming language loop can be made to repeat for each tuple retrieved. With such ability the user at the work station can abort a query and retrieve results a tuple at a time. The tuple at a time retrieval can be useful if GQL is to be used in a browsing mode. The implementation of GQL is adopted to communicate with the INGRES terminal monitor. This has been sufficient to handle the requests from the GQL system's current features.

8.2 Setting Up the Host System Process

Depending on the setup under which the GQL system is used the user at the workstation may have to perform logging in procedures of the host system and set up the process that accepts the queries from the workstation for execution. This may be a simple form filling procedure whereby the GQL system obtains the necessary details

from the user and sends the necessary sequence of commands to the host. Abnormal response from the host during this step cannot be easily handled by the GQL system. Other approach could be to provide a direct communication link between the user at the workstation and the host. This could be a simple line by line communication. The GQL implementation provides a 'Setup Host' menu option which presents a window for direct communication with host. Issuing a 'Open' or 'New' menu options to open an existing local dictionary or initialising a new local dictionary also presents this window to remind the user to set up the host process.

8.3 Local Dictionary Initialisation

The local dictionary of the GQL system must be initialised with the host system schema as the first step in using it. The current implementation consists of modules to do this for the university INGRES database system [Sto76], [ING86]. The following INGRES system relations and their attributes are read by the GQL to initialise the local dictionary.

Relation	**Attributes**	**Format**	
relation	relid	character 12	{relation name}
attributes	attrelid	character 12	{relation for the attribute}
	attname	character 12	{attribute name}
	attfrmt	character 1	{c, i or f for 3 types}
	attfrml	integer	{length of field}

If the host system maintains some relationship information such as that of the entity relationship model these can be extracted and maintained in the GQL system as a set of binary equi-joins.

8.4 Communication Routines

The options available to implement the communication between the GQL and host system are,

1. Using the Macintosh serial line communication port to communicate directly; this can be in a synchronous mode or asynchronous mode.
2. Using a communication network to send and receive packets.

The communication routines used in testing the GQL system with the INGRES database on UNIX machines uses the serial port in a synchronous mode. The requirement for communication between the host and GQL system current version is very simple. The GQL system sends a query to the host system as QUEL statements for processing by the INGRES monitor, then waits for all the result tuples to be received.

8.5 Database Evolution

The information that is built into the local dictionary of GQL will become invalid when the host schema definition is changed. Handling changes in the schema of the host system database has not been fully addressed in the design of the GQL system. The current implementation requires the information to be rebuilt when this occurs. The evolution of the database schema cannot be handled incrementally in most existing database systems. The INGRES system handles evolution by redefining the schema. Thus modules that are dependent on the schema definition have to be modified manually to be consistent with the changed schema. Handling changes to the schema in a structured manner is an issue that is yet to be addressed successfully in relational technology, though some improvements along this line could be expected in the near future.

In the GQL system information layers added to the local dictionary will become invalid when changes to the host system schema are made. Thus, when the host system changes, the local dictionary must be reinitialised and the information layers must be rebuilt. As a means of warning the user the GQL system could maintain a time stamp from the host system and compare it with the current stamp on the host when the GQL system is started up. Ultimately GQL could be integrated into a more comprehensive

system that includes data modelling tools as well, and handle the issue of evolution more satisfactorily.

Since the subquery structure and the local dictionary structure is under the control of a single system, certain types of changes in the host system schema may be handled automatically by the GQL system. One approach could be to generate a new dictionary based on the old dictionary and a set of specifications for the changes. The first requirement to implement this would be to formalise the specification of changes. Some of these could be:

1. attribute evolves to become entity
2. addition of new attribute
3. addition of new entity
4. addition of new relationship
5. deletion of attribute cascades to delete equi-joins, cascades to delete subquery nodes.

8.6 GQL in an Integrated Interface.

GQL must be regarded as a graphical query interface which should be part of an integrated front end to relational databases. Therefore the major effort has gone into developing and implementing a graphical querying methodology. Thus the issues of database evolution and communication must be first resolved as part of the integrated front end before GQL can resolve it within its domain. A data modelling tool that forms part of this front end would be the ideal agent for specifying the evolutionary changes of a database. The communication routines used by the front end could be enhanced to make full use of Macintosh serial port as described in §8.3. The communication routines used in the current version were included to illustrate the viability of GQL.

Conclusion and Future Developments

The principal result of this thesis is the production of a graphical technique to express relational queries that has the following advantages.

1. Using 'direct manipulation' to formulate and edit relational queries. Feed back is given to guide the user to formulate correct queries. Ensure correctness of a query by presenting only valid choices to the user while building query.
2. User can access and select information about the schema with ease. The ability to capture details of associations between relations a manner useful for query formulation is built into GQL. User can select them for use in query from lists that GQL presents.
3. Facility to formulate queries incrementally, using the results of previous query.
4. To preserve reusable parts of query and the ability to use these in building new queries.
5. Ability to view query in a hierarchical levels of details through simple user actions.
6. A graphical language that could be used as a front end to any relational database schema independent of a particular semantic model.

GQL is implemented using readily available hardware, namely the Apple Macintosh, to provide an easy to use interface to relational DBMS which is not available as part of most relational DBMS products or independently. GQL permits queries with many joins to be formulated with ease and also has the ability to retrieve aggregate values. The ability to explode levels of detail is a desirable feature for interactions that involve humans. GQL provides this ability graphically to achieve good results. The power of GQL is adequate for a wide range of retrieval operations. In addition greater power of the host system DML can be accessed from GQL through the host system window.

In §5.12 some suggestions for extensions that fit well into the GQL design have been detailed. The GQL interface can be used as a casual user tool with minimum training and can also serve as a tool for efficiency in the hand of an expert. The ability to build information layers in GQL also has the advantage that it can be local to the workstation and tailored to the users needs. A tool like GQL could also be used as an intermediary in a distributed database environment by providing a uniform front end to differing data models.

DDL and DML operations other than retrieval have not been added to GQL. These operations should only be provided through an integrated system that includes data modelling tools. GQL could be interfaced to such an integrated system, thus obtain some semantic information that it requires from it.

Providing update facility through GQL potentially enables the user to make changes to a database without the user being aware of the consequences of the changes. These are the cascading effect of the changes due to constraints that may be imposed on the schema such as the membership class requirements. However a raw update facility to be used by knowledgeable users can be provided through direct use of the host system process. In the GQL implementation the INGRES monitor is used as the host system process and a user at the work station can enter INGRES monitor commands directly through the window in the GQL system provided to set up the host system process.

A report writing module can be appended to the GQL to produce formatted reports. This can be designed independently and displayed within the GQL query, where associations can be made between the fields in the report and the target fields of the query. To achieve greater functionality some form of control specification and arithmetic capability could be added to this module, in effect creating a fourth generation tool as discussed in the introduction. An achievable step in this direction is to be able to use the query expressed using GQL by translating it into a data manipulation statement that can be embedded in a host system programming language.

The design and implementation of the GQL system has been useful in gaining an insight into the area of semantic modelling. The predicate calculus formalism for expressing logic and the attempt to translate this into a graphical formalism has given useful experience in formal specification of logic. The difficulty in designing a graphical interface in general and a graphical relational query language in particular, which is to aim for the power of a linear language must be mentioned. Human computer interaction studies and user interface considerations were interesting areas which needed to be considered in the implementation of GQL.

Appendix A

Sample Relational Database Schema

The relations in the sample schema used in this thesis are listed below and an ER diagram representing this is given in figure A. This is the same figure given in chapter 4.

Item (number, name, dept, price, qoh, supplier)

Dept (number, name, store, floor, manager)

Sale (number, date, store, dept, item, quantity, employee, credit)

Employee (number, name, salary, manager, birthdate, startdate)

Store (number, city, state)

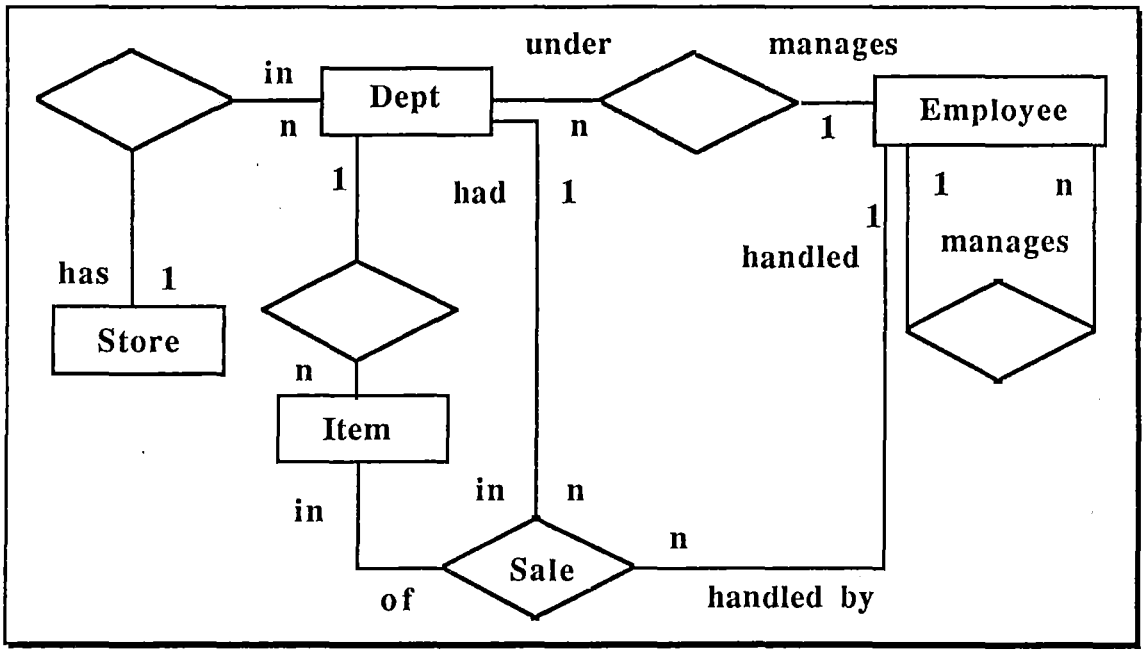


Fig. A

Appendix B

Data Structures

B.1 Use of Surrogates for Dictionary Objects

All dictionary objects in the GQL system are maintained as relocatable objects in the application heap in the Macintosh user work space and referenced by handles. In fact Macintosh tool box [Mac85] permits two ways of maintaining objects in the system heap. In Macintosh terminology the pointers to these two types are called pointers and handles. Handles refer to objects through an extra level of indirection and thus enabling these objects to be relocated by the system during memory compaction without invalidating users handles to these objects. Even though some dictionary objects such as entities and attributes are of a permanent nature and could well have been treated as non relocatable objects this would have required handling two types of dictionary objects. Objects such as subqueries are not permanent in nature, and should be handled as relocatable object if fragmentation of the application heap is to be avoided. Therefore to facilitate a uniform handling of all dictionary objects all objects are treated as relocatable objects.

During execution these objects can be cross referenced using Macintosh Pascal handles to these objects. Maintaining these cross referencing information in the dictionary disk file can be done in one of two ways.

1. Using key values. In this approach the dictionary writing routines must convert the handle information in the application heap into key values which can be written into the disk file. The dictionary reading routine can use these key information to set up the handles.
2. Assign surrogate values to objects that are cross referenced. The dictionary reading and writing routines can be compact and faster under this approach. The Global application area is used to hold arrays of handles. An array is

used for each type of dictionary object to store the handles to the actual objects. The index of the arrays is used as the surrogate value for the dictionary objects.

B.2 Data structures for First and Second Layer of Dictionary Objects

Except for the connection between a Relationship object and the associated Attribute-Group objects of §6.6 all associations between objects in these two layers are one to many in cardinality. Thus all these objects are set up as link lists in the GQL implementation, the owner in the connection holds all the header information about the link list. In the first two layers the following link lists are maintained.

1. Entities to Attributes
2. Domains to Attributes
3. Entities to Attribute-Group-Of-Entities
4. Attribute-Group-Of-Entities to Attribute-Grp-Atoms
5. Entities to Rel-Links

The relationship object requires special handling since it is desirable to maintain these as link lists from the entities involved in the relationship. A relationship may be defined where both entities are the same such as the employee and manager in the sample database in appendix A. An additional rel-link object is added to the dictionary and using this link lists are maintained for the relationship. Each relationship is represented twice in the re-link.

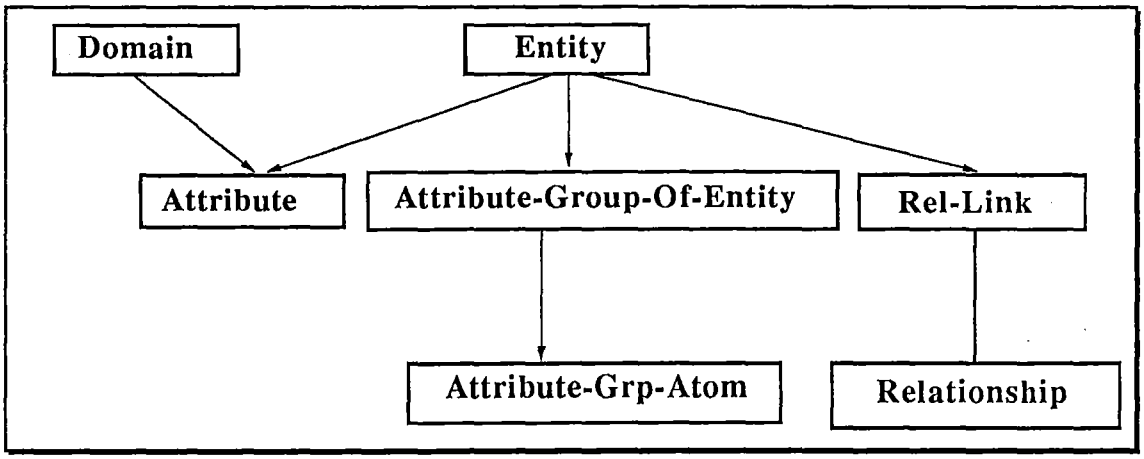


Fig. B.1

The objects in the above lists describe database schema and relationships among the entities of the schema. Thus all the above five link lists are a one off creation and objects once added to the lists are not deleted, removing these objects involve changing the schema itself. This type of link lists contrast to the link lists structure used to express queries as described in §7.3. Thus by using the surrogate values for cross referencing these link lists are stored with their links in the disk file. For permanent dictionary objects surrogate values are assigned in ascending order in the order of appearance. Therefore recreation of the link list is not required during startup time. The Third layer of the dictionary described in the next section differs in this respect from the first two layers. The following Pascal data structures are used in the implementation to represent the above information.

B.3 Data Structures for Query Expression and Third Layer

Objects in this dictionary layer are not allocated surrogate values since these are of a temporary nature. Each subquery is however allocated a surrogate value for referencing by other queries where this is used. This is only a simplification step, since subqueries can easily be identified by the unique names that users give to them. The following link lists are maintained for this layer. These link lists have to be set up every time the dictionary is read in. They are

1. Subquery to Tables.
2. Subquery to Join-Nodes and SubQ-Nodes.
3. Subquery to Theta-Join.
4. Table to Attributes.
5. SubQ-Node to SubQ-Node-Connect.

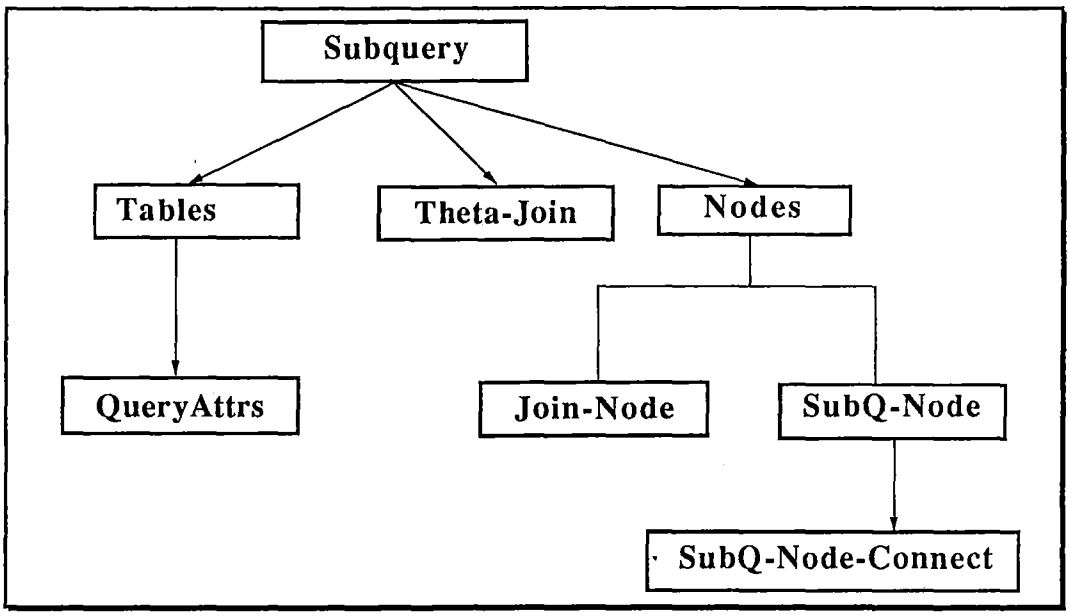


Fig. B.2

B.4 Pascal Data Structures for Dictionary Objects

The Pascal data structures used in the implementation of GQL are listed below. The objects in the GQL implementation can be classified into those that are permanent, and thus given surrogate values, and those that are not. Link lists that involve permanent objects use the following header and link types.

```
LnkHdr = RECORD
          NoOfItem : integer;
          Hdr : integer;                    {Index to global
                                            array for surrogate
                                            allocation}
          Tail : integer;                   {As above}
          END;
```

```
LnkComp = RECORD
          Owner : integer;                    {As above}
          Link : integer;                     {As above}
          END;
```

The object types that use the above header and link blocks are Entity, Attribute, Attribute-Group and Relationship (§5.4.1-§5.4.2). The interconnections maintained by the link lists are shown in figure B.1 and represent the first and second layer of the dictionary. A global array is declared for each of the above object types and the index to this array is the surrogate value assigned to the instance of the object. The object itself resides in the application heap in the Macintosh user area and a pointer to the object is placed in the global array.

```
REntity = RECORD
          Name : Str20;
          AttLnkHdr : LnkHdr;
          RLnkHdr : LnkHdr;
          AGLnkHdr : LnkHdr;
          END;


PtrEntity = ^REntity;
HndEntity = ^PtrEntity;

LEntity = ARRAY[1..ESize] of HndEntity;          { global array for
                                                 surrogate allocation}


RDomain = RECORD
          Name : Str20;
          Tpe : char;
          Lngth : integer;
          AttLnkHdr : LnkHdr
          END;

PtrDomain = ^RDomain;
HndDomain = ^PtrDomain;
LDomain = ARRAY[1..DSize] OF HndDomain;          {global array for
                                                 surrogate allocation}


RAttribute = RECORD
          Name : Str20;
          EntLnkComp : LnkComp;
          DomLnkComp : LnkComp;
          END;

PtrAttribute = ^RAttribute;
HndAttribute = ^PtrAttribute;
LAttribute = ARRAY[1..ASize] OF HndAttribute;    {global arry for
                                                 surrogate allocation}
```

```
RAG = RECORD
          Name : integer;          {Attribute-Group-Of-Entity}
          EntLnkComp : LnkComp;
          AGALnkHdr : LnkHdr;
          IsItKey : boolean;
          END;

PtrAG = ^RAG;
HndAg = ^PtrAG;
LAG = ARRAY[1..AGSize] OF HndAG;                {global array for
                                                surrogate allocation}


RAGA = RECORD                                   {Attribute-Grp-Atom}
          AGLnkComp : LnkComp;
          AttributeNo : integer;
          END;

PtrAGA = ^RAGA;
HndAGA = ^PtrAGA;
LAGA = ARRAY[1..AGASize] OF HndAGA;             {global array for
                                                surrogate allocation}


RRelShip = RECORD                               {Relationship}
          EntOne : integer;
          EntTwo : Integer;
          AGOne : integer;
          AGTwo : integer;
          TextOne : Str20;
          TextTwo : Str20
          END;

PtrRelShip = ^RRelShip;
HndRelShip = ^PtrRelShip;
LRelShip = ARRAY[1..RSize] OF HndRelShip;       {global array for
                                                surrogate allocation}


RRelLnk = RECORD                                {Rel-Link object}
          RelShipIndx : integer;
          FirstOne : Boolean;
          EntLnkComp : LnkComp;
          END;

PtrRelLnk = ^RRelLnk;
HndRelLnk = ^PtrRelLnk;
LRelLnk = ARRAY[1..RLnkSize] OF HndRelLnk;      {global array for
                                                surrogate allocation}
```

The following header and link types are used in the link lists that involve objects that do not have surrogate values assigned to them. All objects in the third layer of the dictionary, except subquery objects do not have surrogate values allocated to them. Therefore no global arrays are declared for these objects. The interconnections maintained using link lists for these objects is shown in figure B.2.

```
LnkHeader = RECORD
              NoOfItem : integer;
              Hdr : Handle;                    {Handle to the header
                                                object for the list}
              Tail : HAndle;                   {Handle to the last
                                                object in the list}
              END;

LnkComponent = RECORD
              Owner : Handle;                  {Handle to the owner
                                                ie.  header of the
                                                list}
              Link : Handle;
              END;

TableAtom = RECORD
              TAtomIndx : integer;
              TAbleRect : Rect;
              TupleID : integer;
              TableEnlarged : Boolean;
              ETableShift : integer;
              Elength : integer;
              Eheight : integer;
              AttLnkHdr : LnkHeader;
              SubQLnkComp : LnkComponent;
              TableCntrl : ControlHandle;
              TupleNAme : Str20;
              END;

PtrTable = ^TableAtom;
HndTable = ^PtrTAble;

AttrAtom = RECORD                              {QueryAttr}
           AttrType : integer;
           AttrPointer : integer;
           DomIndx : integer;
           DispLngth : integer;
           HndQual : StringHandle;
           CompOp : integer;
           TargetSpec : integer;
           TableLnkComp : LnkComponent;
           END;

PtrAttr = ^AttrAtom;
HndAttr = ^PtrAttr;

ExprTerm = RECORD
           Ent : Handle;
           Att : integer;
           END;

ExprAtom = RECORD                              {Theta-Join}
           ExprCntrl : ControlHandle;
           ExprRect : Rect;
           QTerm1 : ExprTerm;
           CompOp : integer;
           QTerm2 : ExprTerm;
           SubQLnkComp : LnkComponent;
           END;

PtrExpr = ^ExprAtom;
HndExpr = ^PtrExpr;
```

```
NodeAtom = RECORD                          {Join-Node or SubQ-Node}
           NodeIndx : integer;
           NodeRect : Rect;
           NodeCntrl : ControlHandle;
           SubQLnkComp : LnkComponent;
           CASE NodeType : integer OF
                  RKnd : (
                                TxtIndx : integer;
                                TableOne : Handle;
                                TableTwo : Handle;
                                );
                                0 : (
                                NodeLstHdr : LnkHeader;
                                );
           END;

PtrNode = ^NodeAtom;
HndNode = ^PtrNode;

NodeElement = RECORD                                    {SubQ-Node-Connect}
              LocTAble : Handle;
              TableIndx : Handle;
              NodeLnkComp : LnkComponent;
              END;

PtrNodeElem = ^NodeElement;
HndNodeElem = ^PtrNodeElem;

SubQAtom = RECORD                                       {Subquery}
           SubQName : Str20;
           QPic : PicHandle;
           UseCount : integer;
           SubQLnkCOmp : LnkComponent;
           TabHdr : LnkHeader;
           ExprHdr : LnkHeader;
           NodeHdr : LnkHeader;
           SubQHdr : LnkHeader;
           Surrogate : integer;
           END;

PtrSubQ = ^SubQAtom;
HndSubQ = ^PtrSubQ;

Query : ARRAY[1..SubQSize] OF Handle;          {global  array  for
                                               surrogate allocation}
```

B.5 Table of Memory Requirement

The calculations for the following table is based on a schema size of 100 entities each having 5 attributes, 100 domains, 100 relationships involving 200 Attribute-groups and 300 Attribute-group-atoms. In addition the calculation for 20 subqueries is also included.Each subquery is assumed to have 5 tables, 25 attributes, 5 relationships or subquery node and 2 expressions.

Object Type	Object Size	Total Number	Total size
Entity	38	100	3.8 Kb
Domains	29	100	2.9 Kb
Attributes	28	500	14 Kb
AttrGroup	13	200	2.6 Kb
AttrGrpAtom	6	300	1.8 Kb
RelShip	48	100	4.8 Kb
RelLnk	7	200	1.4 Kb
Subquery	80	20	1.6 Kb
Tables	61	100	6.1 Kb
Query Attrs	24	500	12 Kb
Nodes	32	100	3.2 Kb
Master pointers	4	3000	12 Kb
Code size	80 Kb	1	80 Kb
Global Arrays			6 Kb
Result Buffer	5 Kb	1	5K

Grand total = 160 Kbytes approximately.

Appendix C

Link List Routines in GQL

The implementation of the GQL system hinges on a number of one to many list structures. A set of routines to handle these objects in lists were written. The following advantages have resulted

1. A sizable reduction in the code size
2. Clarity in code

One disadvantage of a uniform set of routines to handle all types of objects is the reduction in the speed of execution. This however has not resulted in any perceivable slow down of the system. In a GQL type of system the performance as perceived by the user is an adequate measure. Most of Macintosh tool box routines use a similar approach to reduce tool box code size.

```
PROCEDURE   InitComponent (VAR LC : LnkComponent);
```
This is called to initialise the linking component of an object that is in the link list.

```
PROCEDURE   InitHeader (VAR HD : LnkHeader);
```
This is called to initialise the header component of an object that is the head of a link list.

```
FUNCTION  NewObject (ObjType : integer) : Handle;
```
To create a new object of the requested type. The various types used are listed in Appendix B.

```
FUNCTION   GetOwn (ObjType : integer;
                   MembHand : Handle) : Handle;
```
Since each link component consists of a handle to its owner, the routine simply returns the handle to the owner.

FUNCTION GetLinkHead (ObjType : integer;
OwnHand : Handle) : LnkHeader;

Returns the value of the header object as the value of the function.

PROCEDURE SetLinkHead (ObjType : integer;
OwnHand : Handle;
NewHead : LnkHeader);

Sets the link head object to the new value passed in NewHead.

PROCEDURE JumpLink (ObjType : integer;
PrevHand, NextHand : Handle);

Adjusts the link components pointed to by the handles to be the successive ones in the list.

FUNCTION GetFirstLink (ObjType : integer;
OwnHand : Handle) : Handle;

Returns the first object in the list described by the parameters.

FUNCTION NextInLink (ObjType : integer;
theHand : Handle) : Handle;

Returns the next object in the list from the object described by the parameters.

PROCEDURE AppendToLink (ObjType : integer;
OwnHand, MembHAnd : Handle);

Appends the object at the end of the list.

PROCEDURE RemFromLink (ObjType : integer;
MembHand : Handle);

Removes the object from the link list.

Appendix D

Development Environment

D.1 Configuration

The development work for this project was carried out on an Macintosh Plus with one megabyte RAM and the 800K internal disk drive with the Macintosh Light Speed Pascal [LSP86] development system. The University INGRES relational database [ING86] system on the UNIX machines available in the Computer Science department was used as the host database system for GQL.

Initially this set up was sufficient. Some disk swapping was still necessary when the Resource Editor, which is part of the Macintosh development environment [Cher85], was used to design the resources that are required for the GQL interface. When the size of the GQL system grew too large to hold LightSpeed Pascal and GQL sources it was necessary to include an additional external disk drive to the development environment. The work was carried out using the hard disk storage medium available through the 'AppleTalk' network in the department.

Figure C.1 shows the software modules that could have been used for development at the time of writing this thesis. The 'Extender' routines while reducing the effort required to develop an application results in greater code size. The toolbox additions provided in the ROM85 library include the list manager routines. The use of list manager routines for developing GQL was considered. Its use in displaying the expanded table of a query requires displaying list of differing types in a tabular form. Interfacing this requirement with the list manager was found to be as difficult as implementing custom made routines. The instance where use could have been made of list manager routines is to use it in the list dialog of GQL as described in §6.4. Here too since it was intended to be used as a selection dialog in several instances custom

made routines were written. Thus the ROM85 library is not used in the current implementation.

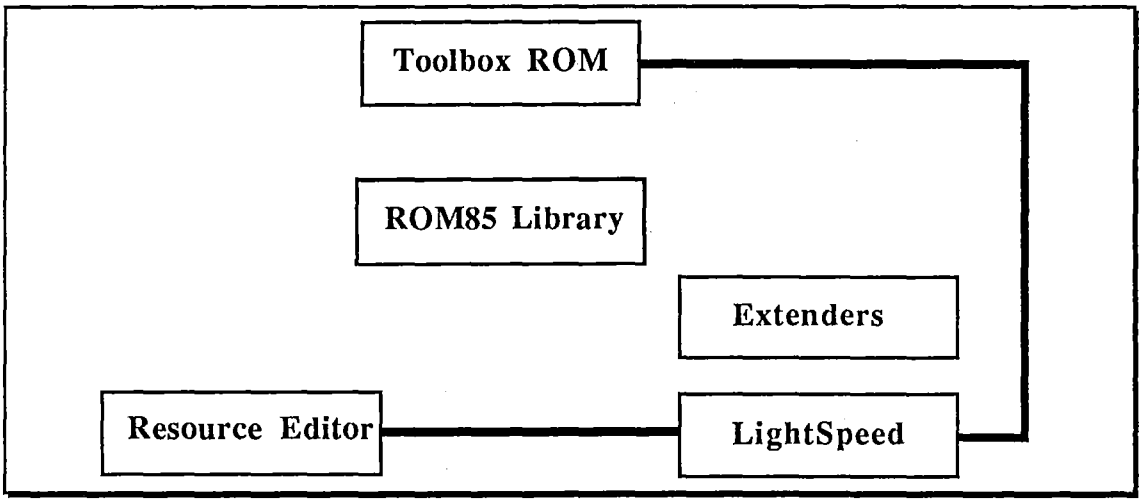


Fig. C.1

D.2 Notes on Experiences with Toolbox

Some justifiable criticism has been levelled at the toolbox regarding its suitability for developing applications by calling these routines. But appearance of additional development aids such as those shown in figure C.1 has addressed some of these. Developing a Macintosh application still requires a lot of experience in programming with the toolbox before productive work can be done. It appears easier to use development tools inevitably introduce extra code and slower execution. This can be alleviated by bigger and faster hardware resources or a new approach to graphical programming such as an object oriented programming language. Object Pascal and MacAPP are two development systems based on object oriented languages that are marketed as Macintosh development system. Their superiority over traditional languages like Pascal for this purpose is yet to be fully tested. The Macintosh 'resource' [Mac85] concept is in fact a step along the line of object orientation in graphical programming.

D.3 Scrolling with Toolbox

In GQL, standard Macintosh scrolling facility using scrollbars is provided for viewing the result display and for viewing theta-joins. The query display itself is not scrollable. The reasons for this are two fold.

An initial design decision was made to provide a non scrollable query display. A scrollable query display does not result in reduction of complexity that is aimed for in GQL. The subquery facility of GQL is a more friendly form of expressing large queries. To keep in line with Macintosh applications scrollable query display was attempted after GQL has been implemented. Technical problems were encountered in this attempt. GQL uses a toolbox defined object called 'controls' for representing its objects. The toolbox scroll bar is also an object of type 'control'. This is the cause of the problem and this can be eliminated by avoiding the use of 'controls' for representing GQL objects.

D.4 Notes on Experience with LightSpeed Pascal

The LightSpeed Pascal system provides some excellent debugging facilities which greatly reduces the number of compilations required. The 'LightsBug', 'Instant' and 'Observe' windows of the system are good debugging tools. One of the flexibility that has been lost as a result of the comprehensive environment is that modules written in another language cannot be interfaced with LightSpeed Pascal routines since LightSpeed system does not generate an intermediate code file that is accessible to outside.

Bibliography

[Ast76] M.M. Astrahan et al., 'System R: Relational Approach to Database Management', ACM TODS, Vol 1, No 2, 1976, pp 97-137.

[Chan86] Shi Kuo Chang, 'Visual Languages: A Tutorial Survey', Lecture Notes in Computer Science Series 282. Visualisation in Programming, 5th Interdisciplinary Workshop in Informatics and Psychology 1986, Ed : P.Gorny and M.J. Tauber

[Chen76] P.P.S. Chen, 'Entity-Relationship Model - Toward a Unified Approach', ACM. TODS, Vol 1, No 1, 1976, pp 9-35.

[Cher85] S. Chernicoff, Macintosh Revealed Vols I and II, Hayden, 1985.

[Codd70] E.F. Codd, 'A Relational Model of Data for Large Shared Data Banks', CACM Vol 13, No 6, 1970, pp 377-387.

[Codd72] E.F. Codd, 'Relational Completeness of Data Sublanguages', Database Systems , Ed. R.Rustin , Prentice Hall, 1972.

[Codd78] E.F. Codd , 'How About Recently', Databases: Improving Usability and Responsiveness, Ed:B.Shneedermann, Academic Press, 1978.

[Codd79] E.F.Codd, 'Extending the Database Relational Model to Capture More Meaning', ACM TODS, Vol 4, No 4, 1979, pp 397-434.

[Date86] C.J. Date, An Introduction to Database Systems, Volume One, Fourth Edition, 1986, Addison-Wesley Inc.

[DBTG71] Data Base Task Group of CODASYL Programming Language Committee, Report. 1971.

[Deyi84] Deyi Li, A PROLOG Database System, Research Studies Press, 1984.

[Fogg84] Fogg D, 'Lessons from a "Living in a Database" Graphical Query Interface', ACM SIGMOD, Rec 14, 1984. pp 100-106.

[Fras86] C.Frasson & M.Er-radi, 'Principles of Icon Based Command Languages', ACM SIGMOD, Rec 15, 1986, International Conference on Management of Data, pp 144-152.

[Gai83] B.R. Gaines, 'Dialog Engineering', Designing for Human-Computer Communication , Ed: M.E.Sime and M.J.Coombs. Academic Press, 1983, pp 23-53.

[Gray84] P.Gray, Logic, Algebra and Databases, Ellis Horwood , 1984.

[Gree78] Greenbatt D et. al, 'A Study of Three Database Query Languages', Databases: Improving Usability and Responsiveness, Ed:B.Schneiderman, Academic press, 1978, pp 77-97.

[Herot82] C.F.Herot, 'Graphical User Interfaces', Human Factors and Interactive Computer Systems, Ed:Yannis Vassiliou, Ablex, 1982.

[Howe83] D.R. Howe, Data Analysis for DataBase Design, Edward Arnold Ltd, 1983.

[Hull87] R.Hull and R. King, 'Semantic Modelling : Survey, Applications and Research Issues.', ACM Computing Surveys, Vol 19, No 3, pp201-260, 1987.

[ING86] J. Kalash et. al., Ingres Reference Manual, University Ingres Documentation.

[Jard76] The ANSI/SPARC DBMS Model: Proc. of the Second SHARE Working Conference on Data Base Management System, April 1976, Ed: D.A.Jardine.

[Jark85] Jarke M et.al., 'Frame Work for Choosing a Database Query Language', ACM Computing Surveys, Vol 17, No 3, 1985, pp 313-340.

[Kent81] W.Kent, 'Consequences of Assuming a Universal Relation', ACM TODS, Vol 6, No 4, 1981, pp 539-556.

[Lar87] J.Larson, 'Graphical Query Languages for Semantic Database Models', Proc. of the AFIPS. National Computer Conference 1987, pp 617-623.

[LSP86] Light Speed Pascal, User Guide and Reference Manual, THINK Technologies, 1986.

[Mac85] Inside Macintosh. Vols I to V.

[Mart85a] J. Martin and C. McClure, Diagramming Techniques for Analysts and Programmers, Prectice-Hall, 1985.

[Mart85b] J.Martin, Fourth Generation Languages. Vol 1, Prentice-HAll, 1985.

[Mar80] F.J.Maryanki and C.S.Roush, 'Definitions of Database Transactions by Casual User', Proc. of the AFIPS. National Computer Conference, 1980, pp 293-300.

[Mat85] S. Matwin and T.Pietreyukowski, 'PROGRAPH: A Preliminary Report', Computer Languages, Vol 10, No 2, 1985, pp 91-126.

[McD75] N. McDonald and M. Stonebraker, 'CUPID- The Friendly Query Language', Proc. of the ACM Pacific Conference, 1975, pp 127-131.

[Mill57] 'The magical number 7±2 : Some Limits on Our Capacity for Processing Information', Psychological Review 63, 1957, pp 81-97.

[Reis75] P. Reisner , R.F. Boyce and D.D. Chamberlin, 'Human Factor Evaluation of Two Database Query Languages', Proc. of the AFIPS. National Computer Conference 1975, pp 439-436.

[Reis77] P.Reisner, 'Use of Psychological Experimentation as an Aid to Development of a Query Language', IEEE Transactions on Software Engineering, SE-3, 1977.

[Schi88] Lecture Notes in Computer Science Series, Advances in Database Technology, Proc. of the International Conference on Extending Database Technology, 1988, Ed: J.W.Schimidt et.al., Springer-Verlag.

[Shne82] B. Shneidermann., 'The Future of Interactive Systems and the Emergence of Direct Manipulation', Human Factors and Interactive Computer Systems, Ed:Yannis Vassiliou, Ablex, 1982.

[Sto76] M.R. Stonebraker, E.Wong, P.Kreps and G.D. Held, 'The Design and Implementation of INGRES', ACM TODS, Vol 1, No 3, 1976.

[Teor86] Teorey,Yang and Fry, 'A Logical Design Methodology for Relational Database Systems Using EER', ACM Computing Surveys, Vol 18, No 2, 1986, pp 197-222.

[Thom83] 'Psychological Issues in the Design of Database Query Languages', Designing for Human-Computer Communication , Ed: M.E.Sime and M.J.Coombs. Academic Press, 1983, pp173-200.

[Ull82] J.D.Ullman, Principles of Database Systems, Second Edition, Computer Science Press, 1982.

[Urs83] P.Ursprung et.al., 'HIQUEL: An Interactive Query Language to Define and Use Hierarchies', Entity-Relationship Approach to Software Engineering, Ed: C.G.Davis et.al, 1983, pp 299-314.

[Webb88] 'A QUEL-to-SQL Data Manipulation Language Translator', Honours Project, Department of Computer Science, University of Canterbury, 1988.

[Zhi83] Zhi-Qian-Zhang et.al., 'Query language for Entity-Relationship Database.', Entity-Relationship approach to Software Engineering, Ed: C.G.Davis et.al, 1983.

[Zlo77] M.M . Zloof, 'Query-by-Example: A Database Language' IBM Systems Journal Vol 16, No 4, pp 324-343.

[Zlo78] M.M.Zloof, 'Design Aspects of Query-By-Example Database Management Language', DATABASES: Improving Usability and Responsiveness , Ed: B. Schneidermann. Academic Press, 1977, pp 99-128